University of Alberta

Personalized Update Monitoring Toolkit Using Continual Queries: System Design and Implementation

by

Wei Tang

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-34426-6

Canada

University of Alberta

Library Release Form

Name of Author: Wei Tang

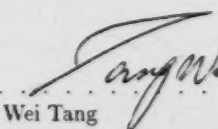
Title of Thesis: Personalized Update Monitoring Toolkit Using Continual Queries: System Design and Implementation

Degree: Master of Science

Year this Degree Granted: 1998

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Wei Tang
615 General Services Building
University of Alberta
Edmonton, Alberta
Canada, T6G 2H1

Date: June 29, 1998

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Personalized Update Monitoring Toolkit Using Continual Queries: System Design and Implementation** submitted by Wei Tang in partial fulfillment of the requirements for the degree of **Master of Science**

Ling Liu

Randy Goebel

Francis Y. Lau

Date: June 23, 1998

Abstract

In rapidly growing open environments such as the Internet, users experience information starvation in the midst of data overload, due to difficulties similar to finding a needle in a haystack. Update monitoring is a promising area of research where we bring the right information to the user at the right time, instead of forcing the user through manual browsing or repeated submission of queries.

In this thesis, we present the design and implementation of an event-driven update monitoring system using continual queries [25, 27]. A Continual Query (CQ) is a standing query that monitors update of interest and returns result whenever the update reaches specified thresholds. Each continual query consists of three components: a query component, a trigger condition, and a stop condition. In contrast to normal queries whose scope are limited to past and present data, the scope of a continual query includes past, present, and *future* data. The main contributions of this thesis is the design and implementation of the first prototype of the continual query system, capable of push-enabled data management and event-driven information delivery. The key components of this prototype system are a continual query (CQ) manager, a trigger condition evaluator, and a set of event detectors. The CQ manager is responsible for coordination of tasks and communications between CQ clients and CQ server and between CQ server and CQ wrappers. The condition evaluator and event detectors are responsible for monitoring updates according to specified update thresholds of interest and the time constraints. The distinct features of this continual query prototype system include: reusing and extending conventional DBMS components, providing push-enabled services by incorporating distributed event-driven triggers, and combining pull and push services in a unified framework.

To my dear parents: Gangdou Tang and Xiangfan Zeng

Acknowledgements

I would like to express my thanks to Dr. Liu, my supervisor, for her invaluable guidance and assistance. She always kept me focused and motivated during my thesis work. I would also like to thank the members of the examining committee, Randy Goebel and Francis Y. Lau for their insightful comments.

I am also grateful for the suggestions and help from team members of the CQ/DIOM project and my friends. Special thanks are given to David Buttler, John Biggs, Wei Han and Tong Zhou.

Finally I would like to thank my parents for their encouragements that helped me to overcome all the new challenges encountered in my study and in my life. It is my parents who helped me to become strong physically and spiritually.

Contents

1 Introduction 1
- 1.1 Thesis Motivation . . . 1
- 1.2 Why aren't pull-based systems sufficient? . . . 2
- 1.3 What can a push-enabled system offer? . . . 3
- 1.4 Update Monitoring using Continual Queries . . . 3
- 1.5 Scope and Organization of this thesis . . . 4

2 Continual Query Concept 7
- 2.1 Continual Query Concept . . . 7
- 2.2 Continual Query Examples . . . 9
- 2.3 Continual Queries v.s. ECA Rules . . . 10
- 2.4 Continual Query Specification . . . 11
  - 2.4.1 Specification Semantics . . . 11
  - 2.4.2 Specification Syntax . . . 12

3 System Architecture 16
- 3.1 Overview of the Mediator/Wrapper Concept . . . 16
- 3.2 Overview of the CQ System . . . 18
- 3.3 Information Integration in CQ - DIOM . . . 22
- 3.4 Client-Server Design . . . 25

4 Continual Query Execution Model 26
- 4.1 A Quick Look at Continual Query Execution . . . 26
- 4.2 Continual Query Execution Model . . . 28
  - 4.2.1 Basic Coupling Modes . . . 28
  - 4.2.2 Continual Query Installation . . . 31
  - 4.2.3 Event Detection . . . 32
  - 4.2.4 Condition Evaluation . . . 34
  - 4.2.5 Issues on Efficient Condition Evaluation . . . 36
- 4.3 Performance Evaluation Issues . . . 37

5 Differential Evaluation Algorithm for Continual Queries 38
- 5.1 Motivation . . . 38
- 5.2 Notations and Terminology . . . 40
  - 5.2.1 Differential Relations . . . 41
  - 5.2.2 Basic Operations . . . 42
- 5.3 Differential Evaluation of Continual Queries . . . 43
  - 5.3.1 Computing the Differential Results for Continual Queries . . . 44
  - 5.3.2 Optimization based on Differential Operators . . . 46
  - 5.3.3 The Differential Re-evaluation Algorithm . . . 50
- 5.4 Processing Continual Queries: Simple Examples . . . 52
- 5.5 Discussion . . . 53
  - 5.5.1 Strawman Performance Arguments . . . 53

5.5.2 Query Refinement . . . 54
5.5.3 **Garbage Collection of Differential Relations** . . . 55
5.5.4 Generation of Delta Relations . . . 55
5.6 Implementation Consideration for DRA . . . 55

6 Prototype Design and Implementation **59**
6.1 System Requirements Analysis . . . 59
6.1.1 Analysis of Non-Functional Requirements . . . 59
6.1.2 Analysis of Functional System Requirements . . . 61
6.1.3 Functional Components . . . 63
6.2 System Design . . . 65
6.2.1 UIP components . . . 66
6.2.2 IP components . . . 76
6.2.3 CQP components . . . 77
6.2.4 OP components . . . 83
6.2.5 MST components . . . 84
6.3 Coding Design . . . 85
6.3.1 Perl Programming Language . . . 86
6.3.2 Common Gateway Interface . . . 86
6.3.3 Dynamic HTML . . . 87
6.3.4 Online Database Accessing . . . 88
6.3.5 Emerging Technologies . . . 90
6.3.6 Online Live Demo of the Prototype . . . 91
6.4 User Interface Walkthrough . . . 91
6.4.1 Main Menu Window . . . 91
6.4.2 Client Services . . . 91
6.4.3 Administration Services . . . 102
6.5 Further Discussions . . . 103

7 Discussion and Related work **109**
7.1 Pull, Push, and Continual Queries . . . 109
7.1.1 Overview of Data Delivery Protocols . . . 109
7.1.2 Overview of Data Delivery Modes . . . 110
7.1.3 Pure Push versus Continual Queries . . . 112
7.2 Related Work on DB Areas . . . 113
7.2.1 Active Databases . . . 113
7.2.2 Materialized Views . . . 113
7.2.3 Commercial database triggers . . . 114
7.3 Related Work in Web-based Systems . . . 114

8 Conclusion and Future Work **116**
8.1 Summary and Conclusion . . . 116
8.2 Future Development . . . 117

Bibliography **119**

A Continual Query Syntax **122**

List of Figures

3-1 The cooperation network of mediators and wrappers 17
3-2 A Sketch of the CQ System architecture 19
3-3 DIOM System Architecture 23
3-4 A Sketch of the Distributed query scheduling framework in DIOM 24
3-5 Example Scenarios of the Client/Server coordination among CQ components 25

5-1 Delta Table Auto-generation Perl Code 58

6-1 Top-level Architecture Flow Diagram of *Continual Query System* 64
6-2 UIP components 66
6-3 Truth Tables for *Event Relational Operators* 74
6-4 Wrapper Components 82
6-5 Sample Continual Query Result Page 85
6-6 The CQ Services Main Window 92
6-7 User Registration Form 93
6-8 Choose Data Source Screen 94
6-9 Normal Query Entry Form Screen 95
6-10 A sample result page for Normal Query 96
6-11 CQ Installation Step 1 screen 97
6-12 CQ Installation Step 2 (time-based) - Query Construction 98
6-13 CQ Installation Step 2 (time-based) - Trigger and Stop Condition 99
6-14 CQ Installation and Execution Log Information 100
6-15 CQ Installation Step 2 (content-based) - Trigger and Stop Condition 101
6-16 User Login Screen for Browsing CQ 102
6-17 View Installed CQ Screen 103
6-18 Source Meta Query Screen 104
6-19 Data Source Login Screen 105
6-20 Table and Action Choice Screen 105
6-21 Data Update Screen 106
6-22 Continual Query Weather Watch source front page 107
6-23 Continual Query Installation(query input) for Weather Watch data source 108

7-1 The data delivery flow in a broadcast-based push service 112
7-2 The data delivery flow in a broadcast-based push service 112

Chapter 1

Introduction

1.1 Thesis Motivation

The World Wide Web (usually referred to as *the Web*) has made an enormous amount of data freely accessible over the Internet. However, finding the right information in the midst of this mountain of data has been likened to finding the proverbial needle in a haystack. This phenomenon has been called "information starvation due to data overload."[1] Commonly used search engines, including web robots (e.g., AltaVista) and indexers (e.g., Yahoo!), have ameliorated the situation somewhat, but the current exponential growth of the Web is quickly aggravating the fundamental problem.

We divide the problem of finding relevant information into two parts. The first part is the search for historical data in the Web. Given its static nature, historical data is best suited for search engines, and generally speaking, data warehousing tools. The second part of the problem is *update monitoring*, which deals with the new information arriving into the Web and the databases. There are many applications in both parts. Let us consider a simple example in decision support systems. On one hand, historical data is used in long term projections and planning, for example, by Wal-Mart in the selection of inventory. On the other hand, update monitoring is used in (near) real-time decisions, for example, by investment bankers in the buying and selling of stock.

While both historical data and update monitoring have interesting applications and research challenges, we focus on update monitoring in this thesis. There are three reasons for this bias. First, managing read-only historical data is a more mature area, with many commercial data warehouse systems available. Consequently, many of the most obvious research questions have been answered. Second, many application domains today have the need for tracking changes in local and remote data sources (e.g., databases, HTML Web pages, unstructured files) and notifying changes if some condition over the data sources is met. Wasteful polling by the users or applications can be avoided, if the data management system monitors the update events of interest (e.g., changes to the relevant object classes or instances) on behalf of the users or applications, evaluates the condition only when a potentially interesting change occurs, and issues planned queries (or actions) as well as change

[1] Gio Wiederhold of Stanford University seems to have been among the first to use this phrase.

notification alerts whenever the update events of interest are signaled. Third, update monitoring introduces special difficulties when heterogeneous data sources (e.g., from Web pages and relational databases) are being monitored together. As a result, update monitoring presents some interesting new research challenges.

1.2 Why aren't pull-based systems sufficient?

Conventional data intensive systems (e.g., DBMSs) are pull-based data delivery systems: queries or transactions are executed only when explicitly requested to do so by a user or an application program. Most current web search engines (such as Altavista, Infoseek, Lycos, Excite, to name a few) are also pull-based, passive information delivery systems, namely the transfer of data from servers to clients is initiated by an explicit client pull. Many applications, such as situation assessment, office workflow control, process control, battle management, which require timely response to critical situations, are not well served by these passive pull-based data management systems. For these time-constrained applications, it is important to monitor events occurring on states of databases, and whenever the updates reach some specified thresholds or satisfying some timing constraints, specific actions (such as queries) will be invoked. For example, inventory control in an automated factory or logistics application requires that the quantity on hand of each item be monitored. If the quantity on hand falls below a specified threshold for some item, then a notification procedure may have to be initiated either immediately or at the end of the working day. Similarly, a situation assessment application requires that various targets be tracked. If one is discovered to be within a critical distance, then the alter code may have to be displayed on the commander's screen with the highest possible priority.

This is also true for many applications using web-based pure pull information delivery systems. With the ongoing advance in World Wide Web (WWW) technology, everyone can publish information on the web independently at any time. On one hand, the flexibility and autonomy of producing and sharing information on WWW is phenomenal. On the other hand, it becomes increasingly difficult when using pure pull system as a solution to deal with the daunting challenge of both navigating, collecting, processing, and tracking data in this dynamic and open information universe. The problem is aggravated when source information changes constantly, but unpredictably. As a result, users have to frequently poll the web sites of interest and fuse the newly updated information manually to keep track of changes of interest, which is a great pain.

As more aspects of business and commerce migrate online, push-enabled data management systems becomes increasingly important because it offers system-supported update monitoring and event-driven information delivery, and it provides timely response (or alert) to critical situations, while reducing the time users spend hunting for the updated information and avoiding unnecessary traffic on the net.

1.3 What can a push-enabled system offer?

Push-enabled data management systems are *event-driven* and *time-critical*: users specify to the system the information they need (such as the events or the update thresholds they are interested in). Whenever the information of interest is available, the system immediately delivers it to the relevant users; otherwise, the system continually monitors the arrival of the desired information and pushes it to the relevant users as it meets the specified update thresholds. In contrast to pull-based systems, the transfer of data from servers to clients in push-based data delivery systems is initiated by a server push in the absence of explicit request from clients. In addition to the difference in data delivery mode (push vs. pull), push-based systems differ from pull-based systems also in data delivery protocol and scope of a query.

Most pull-based systems use *Request/Response* protocol where clients send their request to servers asking information of interest, and servers respond to the request of clients by delivering the information requested; whereas push-based systems use either *Publish/Subscribe* protocol or *Broadcast* protocol [2, 11]. The former delivers information based on the principle that servers publish information online, and clients subscribe to the information of interest; whereas the latter delivers information to clients periodically. Clients who require access to a data item need to wait until the item appears. Furthermore, in a conventional pull-based data management system (either DBMSs or web search systems), the scope of a query is limited to past and present data, whereas the scope of a query in a push-enabled continual query system includes past, present, and *future* data. For example, a query "*tell me the quantity on hand of items that have arrived*" is a typical query in a pull-based DBMS, which is defined over the list of items and their quantity on hand, which have arrived up to the moment when this query is issued; whereas a similar query "*report to me the quantity on hand of items changes every day at 10:00am*" is a continual query in a push-enabled system, the scope of this query covers the items and their quantity on hand at the installation time of this continual query, plus the items whose quantities on hand may change as well as new items that may arrive in the future.

1.4 Update Monitoring using Continual Queries

we have described the problem of update monitoring in open environments such as the Internet. By update monitoring we mean the timely delivery of new information (updates) to users. This is a challenging research problem because of several factors. First, it combines update processing (detection) and query processing (new information filtering). Second, the detection and synchronization of updates in several heterogeneous data sources presents fresh problems of its own.

The Continual Queries (CQ) project is an example of research work being done in the update monitoring area, aiming at developing techniques and software toolkit for update monitoring and event-driven information delivery on the Internet. In contrast to conventional database queries, a

continual query is a combination of a normal query, a trigger condition and a termination condition. The continual execution cycle of the query starts at the continual query installation time and stops when its termination condition becomes true. During the continual execution cycle, whenever updates to the data sources satisfy the specified trigger condition, the query is conceptually re-evaluated and new results returned to the user or the application that issued the query. The trigger condition may be time event, e.g., every Monday at 8am, or content-based event, such as "Microsoft's stock price going up by 10%". The query can be either SQL-like queries for database sources or keyword-based search for Web pages. We say that the query is conceptually re-evaluated because of the variety of algorithms and approaches to the standing query re-evaluation.

For each continual query, an update monitoring program (CQ robot for short) creates distributed programs that act together as an intelligent assistant, keep track of information sources that are available (on the Web and elsewhere), how to access them, and the changes that happened. Whenever updates at the data sources reach a specific update threshold or a timed event, the CQ robot computes and integrates the new results and presents them to the user. Compared with the pure pull (such as DBMSs, various web search engines) and pure push (such as Pointcast, Marimba, Broadcast disks [1]) technology, the CQ project can be seen as a hybrid approach that combines the pull and push models by supporting personalized update monitoring through an integrated client-pull and server-push paradigm.

1.5 Scope and Organization of this thesis

This thesis presents the design and implementation of the first prototype system that support push-enabled data management and event-driven information delivery in a distributed and open environment such as the Internet and intranets. The theoretical model and the architecture for specification and execution of continual queries are developed based on the previous result of the CQ [27, 25] project and the DIOM [26, 22] project. This prototype illustrates the ability to automate the update monitoring process using the continual query concept with the ability to allow users to install both time-based event triggers and content-based event triggers.

The main contribution of this thesis project is the systematic development and implementation of three-tier architecture for building a continual query system, especially the experiment in constructing event-driven mediators and continual query enabled wrappers. Rather than starting from scratch, the CQ prototype system development takes advantages of the conventional pull-based DBMS services to the extent possible and adds push-based data delivery elements to them when it is necessary. For instance, rather than introducing a new event-driven query language, we use continual queries to express trigger conditions and unifies them with the regular SQL queries so that continual queries may be expressed in SQL with minimal additions to the language. We provide facilities for monitoring of user-defined operations such as decrease by percentage or value and increase by percentage or value. Such operations allow users to model the event '*stock price drop by*

%10 defined over the data sources or abstract views, in addition to lower level database operations like SQL INSERT, DELETE, and UPDATE. More concretely, the design and implementation of the *continual query* prototype system addresses the following issues:

- The design of the continual query specification language, including the specification of time events and content-based events, and the specification of continual query trigger condition and stop condition.
- The implementation of a subset of the proposed syntax with an easy-to-use graphical user interface.
- The design and implementation of the execution model for efficient processing of continual queries, including the detection of base events such as updates on the source data of interest, and the detection of composite events such as those events that are composed by means of logical AND and/or logical OR as well as system built-in event operators.
- The implementation of interactive interface programs that allow users to install continual queries, browse installed continual queries, trace the execution process of continual queries, and test components of the prototype.
- The experiment of the continual query server kernel components such as event detectors and trigger condition evaluator with different types of CQ wrappers such as the CQ weather watch wrapper (HTML data sources) and the CQ bibliography information service (flat `.bib` files).

The first CQ system prototype chooses to implement the user interface as WWW application. The main technologies used in the prototype implementation include CGI scripts for the main modules of the CQ server and Java applets and servlets with the use of JavaScript for some portions of the GUI. We experiment with the prototype implementation with four different types of data sources:

- An Oracle database which is remotely accessible through SQL, OraPERL, and SQLNet.
- An SQL database which is remotely accessible through JDBC.
- A collection of UNIX files which are accessible through a Perl script or a Java applet.
- An HTML source which is accessible through our semi-structured information wrapper and filter utility.

Although all four sources support different access methods, the wrappers hide all source specific details from the application/end-users. CQ users may pose a query on the fly, and install the query as a continual query by specifying the interested update threshold using the CQ trigger and specifying the continual duration using the CQ termination condition. We also provide a testbed

which consists of a simple interface to allow users to experiment the updates at the data sources and watch the CQ system to compute the update threshold and evaluate the trigger, and alert or notify the user by email the new updates that match the query. In addition, we offer a set of client and system administration services such as browsing and updating the installed continual queries, cancelling some running continual queries upon request, and tracing the CQ trigger evaluation status and the update monitor status.

The rest of the thesis is organized as follows: We present the concept of continual queries and the continual query specification language in Chapter 2. We illustrate the continual query (CQ) specification, in particular CQ trigger definitions, through a number of examples. In Chapter 3 we overview the architecture of the continual query system and its extension to the conventional DBMSs and pure pull web search systems. We discuss the execution model of continual queries in Chapter 4, including event detection and condition evaluation and different communication protocols between triggering transactions (update transactions) and triggered queries and alert messages (such as sending email notification, firing a subsequent execution of a query). These protocols range from synchronous or asynchronous cooperation, causally dependent or independent scheduling, deferred or immediate notification, to execution of triggered actions in a same or separate transaction as the triggering event. Chapter 5 reviews the idea and algorithm for differential processing of continual queries, and reports the implementation design consideration of the differential re-evaluation algorithm (DRA). The system design and implementation issues are covered in Chapter 6. We discuss the issues on information delivery in general and related work in Chapter 7 and summarize the contributions of this thesis and directions for future work in Chapter 8.

Chapter 2

Continual Query Concept

Continual queries are standing queries that monitor updates and return results whenever the updates have reached specified thresholds. A continual query consists of three key components: query, trigger, and stop condition. In contrast to ad-hoc queries in conventional DBMSs or web search engines or query systems, a continual query, issued once, runs continually over the set of information sources. Whenever its trigger condition becomes true, the new result since the previous execution of the query will be returned. The trigger part of a continual query specifies events or situations to be monitored. We distinguish primitive events from conditional (logical) events and allow events to be composed of other events. We use primitive events to model basic database operations (such as INSERT, DELETE, UPDATE), basic time events (such as at time-specification, every time-period, and after time-period), or signals from arbitrary processes. We use conditional events to model various conditional situations to be monitored. We provide a rich set of event composition operators (such as logic operators: conjunction, disjunction, negation; and execution dependency operators: serial, serial alternative, parallel, parallel alternative) to support composition of events.

Continual queries are useful both to external applications and as a convenient mechanism for implementing push-based data delivery functions beyond conventional storage, retrieval, and update of data in conventional DBMSs. Some examples of pull-based functionality can be implemented in a unified way using continual queries and are described later in this chapter.

2.1 Continual Query Concept

A continual query is defined by a triple (Q, T_{cq}, `Stop`), consisting of a normal query Q (e.g., written in SQL), a trigger condition T_{cq}, and a termination condition `Stop`. The initial execution of a continual query is performed as soon as it is installed. The first run of Q is performed over the past and present data at the data source and the whole result is returned to the user. The subsequent executions of Q are performed whenever a new update event occurs (is *signaled*) and the trigger condition T_{cq} becomes true. For each execution of Q, only the new query matches since the previous execution are returned to the user unless specified otherwise. Thus continual queries are defined

over past, present, and future data, whereas the domain of pull queries is limited to past and present data.

Continual Semantics. Let us denote the result of running query Q on database state S_i as $Q(S_i)$. We define the result of running a continual query CQ as a sequence of query answers $\{Q(S_1), Q(S_2), \ldots, Q(S_n)\}$ obtained by running query Q on the sequence of database states $S_i, 1 \leq i \leq n$, at each given state S_i $(i > 0)$, $Q(S_i)$ is triggered by $T_{cq} \wedge \neg \texttt{Stop}$.

The basic events that cause continual queries to fire may be standard database operations such as INSERT, DELETE, UPDATE, or the events that cause clock signals (e.g., check the balance of all bank accounts at *5:00pm everyday*), or any user- or application-generated signals (e.g., a failure signal from a diagnostic routine on a hardware component). Furthermore, the trigger conditions to be monitored may be complex, and may be defined not only on single data values or individual database states, but also on sets of data objects (e.g., the total of orders of items exceeds the current inventory level), transitions between states (e.g., the new position of the ship is closer to the destination than the old position), trends and historical data (e.g., the output of the sensor increased monotonically over the last two hours).

We support two types of trigger conditions: time-based trigger condition and content-based trigger condition. Three types of temporal events are supported for *time-based* trigger condition:

1. absolute points in time, defined by the system clock

   For example, 7:30:00 pm., March 30, 1998.

2. regular or irregular time interval

   For example, every Monday or every two weeks (regular) or every first day of the month (irregular)

3. relative temporal event

   For example, 50 seconds after event A occurred

A *content-based* trigger condition can be expressed in terms of a database query, a built-in situation assessment function, or a user-defined method. Examples include: a simple condition on the database state (e.g., execute Q whenever a deposit of $10,000 is made), an aggregate function on the database state (e.g., execute Q when a total of 1 million dollars of deposits have been made, or execute Q when the stock price of Microsoft drops 5%), and a relationship between a previous query result and the current database state (e.g., execute Q when a total of 1 million dollars of deposits have been made since the previous execution of Q). One extreme case of content-based trigger is immediate: report to me whenever a change to the source data occurs. In addition, composite events made up from these primitive events (e.g., the serial sequence of two events: event B occurs after event A) are also supported.

The **Stop** condition specifies the termination condition of a continual query. **Stop** conditions can be specified in terms of time-based or content-based event expressions. Both the trigger condition T_{cq} and the termination condition **Stop** are evaluated prior to each subsequent execution of the query component Q.

2.2 Continual Query Examples

We below provide two continual query examples, the first one uses time-based triggering event and the second one uses content-based triggering event.

Example 1 Given a continual query *"report to the manager every day at 6:00pm all the banking activities of the day for those customers whose total withdraws reach $2,000"*. It is expressed as follows:

```
Create CQ banking_activity_watch as
Query:
    SELECT customer_id, account_no, withdraw_amount
    FROM   Account
    GROUP BY customer_id having SUM(withdraw_amount) > 2000;
Trigger: 6:00pm everyday
Stop: 1 year (by default)
```

The trigger condition is specified by a regular time interval (everyday) and a starting time point (6:00pm).

Example 2 Suppose we have a continual query installation request *"notify me in the next six months whenever the total of quantity on hand and quantity on order of items drops below their threshold"*. This request is captured by the following continual query expression:

```
Create CQ inventory_monitoring as
Query:
    SELECT item_name, item_no, qty_on_hand, qty_on_order, threshold
    FROM Item_Inventory;
Trigger:
    qty_on_hand + qty_on_order < threshold;
Stop:  six months
```

Here are some other examples of continual queries: "tell me the flight number whenever a plane has been in this sector for more than 5 minutes", "notify me whenever IBM stock price rises by 5%", or "report to me the most recent transportation plan between port of Savannah and Fort Stewart Military Reservation, whenever there is snow, heavy rain, or any other unexpected weather changes in that region".

2.3 Continual Queries v.s. ECA Rules

Continual queries, at the first glance, may seem to bear resemblance to ECA rules in active databases [6, 30]. One might view continual queries as a subset of ECA rules. However, they are quite different not only in functionality coverage and usage perspective but also in execution model and implementation architecture. In this section we briefly discuss the differences between continual queries and ECA rules.

First, update events in ECA rules are explicitly specified by the users, whereas update events in continual queries are implicitly implied in the trigger condition, and derived by the system during the installation phase of the continual queries. Recall Example 2 given in Section 2.2, at the installation phase of the continual query **inventory_monitoring**, the update events identification module identifies three basic update events to be relevant to the trigger condition of the given continual query. They are **UPDATE qty_on_hand(item)**, **UPDATE qty_on_order(item)**, **UPDATE threshold(item)** for **item** in **Item_Inventory**. This means that any update on **qty_on_hand**, **qty_on_order** or **threshold** will signal the evaluation of the trigger condition "qty_on_hand + qty_on_order < threshold". However, using ECA rules, one may specify the follow rule:

```
Event: Update qty_on_hand(item)
Condition: qty_on_hand(item) + qty_on_order(item) < threshold(item)
Action: submit_order(item)
```

Note that this ECA rule has the same trigger condition as the continual query in Example 2. However, this rule means that the condition is evaluated only when the update on **qty_on_hand** occurs, even though the updates on **qty_on_order** or **threshold** may equally be possible to cause the violation of the trigger condition "**qty_on_hand + qty_on_order < threshold**". In short, ECA rules provide flexibility to allow users to explicitly specify what update events are of interest at will, rather than restricting the update events to those that have direct impact on the trigger condition within the same rule. Such flexibility, however, may results in passing over the situations that should be alerted according to the trigger condition.

Second, continual queries require explicit specification of termination condition. In the absence of **Stop** condition, a system default value (such as one year) will be used. Introducing termination condition as a necessary component of continual queries guarantees that alerts or update reports will send only to the right users at right time or under the specific constraints. While ECA rules terminate a rule execution by requiring users to manually delete the rule from the rule base. No system controlled termination is provided. We consider the support for system-controlled termination condition as a desirable and practical capability for a push-enabled active data management system.

Thirdly, although situation monitoring is one of the canonical applications of ECA rules, they are designed as building blocks for general purpose active database systems or production rule systems in

centralized data management systems, whereas continual queries are specifically designed for update monitoring in distributed push-enabled data management systems. Continual queries emphasize on effective and specialized support for personalized update monitoring. Continual queries can be seen as an interesting and effective type of specialization to the ECA rules, which aims at providing more efficient and effective support for personalized update monitoring in a distributed open environment.

Last but not least, actions in ECA rules can be update events which may in turn trigger the same rule again directly or indirectly (i.e., the cascading effects of rules); whereas actions fired, when the trigger condition of a continual query is evaluated to be true, are restricted to the execution of the same query expression, the change notification functions, and the methods to compute the differential result. These actions are side-effect free actions with respect to the data set over which the trigger condition and query component are defined. This feature simplifies many complex issues in ECA Rules, especially those related to the consistency and concurrency issues in advanced transaction management. Such simplification allows us to focus on addressing the issues that are specific to update monitoring and push-based information delivery. We provide a further discussion on active databases and other related work in Chapter 7.

2.4 Continual Query Specification

2.4.1 Specification Semantics

Continual queries, like all other forms of data, are treated as first class objects. There is a continual query entity type, and every continual query is an instance of this type. The difference between continual query entity type and other entity types is that the CQ system understands the semantics of continual queries and invokes a particular operation – *fire* automatically. The functions that define the key components of the structure of a continual query are:

- **Continual query identifier** (cqid). Like any other entity, each continual query (CQ) has a unique entity identifier. Such identifier is generated by the system after the installation of the CQ is successful and the first run of the CQ is fired.
- **Continual query name.** This is a user-defined and optional attribute.
- **Trigger** condition. The trigger component specifies the event that causes the CQ system to fire the subsequent executions of the continual query (CQ). Parameters may be defined for the event; these parameters are bound to actual arguments when the next execution of this CQ is *fired*.
- **Stop** condition. The `Stop` condition specifies the termination semantics of the continual query. It is described by an event expression. Both time-based events and content-based events can be used.

- **Query** component. The query is one of the side-effect free action to be executed when the trigger condition is evaluated to be true and the `Stop` condition is not met. The execution coupling mode between the trigger condition and the query action can be specified explicitly at the continual query installation time to override the system default (see Section 4.2 for further detail).

Both `Trigger` and `Stop` conditions are specified in terms of event expressions. We distinguish between primitive events, composite events, and conditional events. A primitive event is either a basic update event (such as `UPDATE qty_on_hand(item)`) or a temporal event (such as every Monday, 9:00:00pm, March 2, 1998). A conditional event is a conjunction or a disjunction of events, of which at least one of the component events is a conditional event. An atomic conditional event is an event of the form `attribute_name <comparison_op> value`, such as "`stock.price > 100`". A composite event is defined by an event composition expression following the BNF syntax below:

```
<composite_event> ::= <element_event> <event_op> <composite_event>
<element_event>   ::= <primitive_event> | <atomic_conditional_event>
<primitive_event> ::= <basic_update_event> | <temporal_event>
<atomic_conditional_event> ::= <attribute_name> <comparison_op> <value>
<conditional_event>  ::= <atomic_conditional_event>
                         <logic_op> <conditional_event>
<basic_update_event> ::= <db_operations> | <external_signals>
<logic_op>       ::= CONJUCTION | DISJUNCTION | NOT
<comparison_op>  ::= <string_op> | <arithmetic_op>
                     | <built_in_op> | <user_defined_op>
<temporal_event> ::= <absolute_time> | <regular_interval>
                     | <irregular_interval> | <relative_time>
<db_operations>  ::= UPDATE | INSERT | DELETE
<event_op> ::= <logic_op> | <user_defined_op> | <system_built_in_op>
```

A complete BNF description of the CQ system event specification language and the formal semantics of continual query speciation model, including the specification of primitive and composite events, and the algorithm for decomposing the trigger condition components into basic update events and conditional events, are beyond the scope of this thesis.

2.4.2 Specification Syntax

Syntactically, continual queries are defined by specifying trigger condition components in specialized SQL-like `FROM` and `WHERE` clauses plus some special operators, by specifying `Stop` condition in temporal event expressions, and by specifying query components in the SQL-like `SELECT-FROM-WHERE` clauses. Users may give each of their continual queries a meaningful name (such as the continual

query name **banking_activity_sentinel** in Example 1). Continual queries may be defined across over a set of data sources that are autonomous and possibly heterogeneous in nature. These data sources may be structured, semi-structured, or unstructured. Mediators and wrappers are used to decompose the query or trigger condition according to the number of data sources used to evaluate the query or the trigger condition. Details for distribution aspect of the query processing and trigger condition evaluation are beyond the scope of this thesis, interested readers may look at [26].

The following is a fragment of the BNF syntax of the continual query specification language. A more detailed syntax is provided in Appendix A.

```
<CQ>     ::= <Query> <TriggerCond> <StopCond>
<Query>  ::= SELECT <SelectList>
             FROM <ObjectList>
             [WHERE <SearchCondition>]
             [GROUP BY <AttributeList>]
             [ORDER BY <SortSpecList>]
<TriggerCond>    ::= <TimeTriCond> | <ContentTriCond>
<StopCond>       ::= <Month> '-' <Day> '-' <Year> ' '
                     <Hour> ':' <Min> ' ' <TimeZone>
<TimeTriCond>    ::= <MinExpr> '&&' <HourExpr> '&&' <DayOfMonExpr> '&&'
                     <MonthExpr> '&&' <DayOfWeekExpr>
<ContentTriCond>::= <ContTriGroup> | <ContTriGroup>
                     <EventOp> <ContentTriCond>
<ContTriGroup>   ::= <ContPrimitive> <GrpConstraint>
<ContPrimitive>  ::= [<AggreFunc>(] <ObjectName>.<Attribute>[)]
                     <ContTriCondOp> [<Value>]
<GrpConstraint>  ::= WHERE <ContPrimitiveList> [GROUPBY <AttributeList>]
<ContPrimitiveList> ::= <ContPrimitive> | <ContPrimitiveList>
                          <GrpJointOp> <ContPrimitive>
<EventOp>        ::= AND | OR | <sequence> | <parallel>
<ContTriCondOp>  ::= <> | = | < | > | <= | >= | CHANGES | CONTAINS | LIKE
                     | INCBY | DECBY | INCBYP | DECBYP
<AggreFunc>      ::= AVG | COUNT | MAX | MIN | SUM
<GrpJointOp>     ::= AND | OR
```

We provide some examples of continual queries written in SQL-like expression enhanced with user-defined or system built-in functions. We first define a continual query **weather_watch** that monitors weather condition updates in the region from port of Savannah in Georgia to Fort Stewart Military

Reservation every 20 minutes and send mail to Todd using the function **send_mail** whenever the specified update event on weather condition is detected. Suppose that this continual query is defined over a semi-structured data source – the national weather services center website (**www.nws.nova.gov**), and the continual query name is specified in the **Create CQ** clause. The trigger condition is specified in the **Trigger** clause, the termination condition is specified in the **Stop** clause, and the query component is specified in the **Query** clause. Here is the specification of this continual query:

```
Creat CQ Savannah_weather_watch as
Query:   SELECT *
         FROM www.wns.nova.gov
         WHERE location like 'Savannah' AND state = 'Georgia';
               OR location like 'Fort Stewart';
Trigger: every 20 minutes;
Stop:    1 year (default).
```

This continual query specifies the request for monitoring updates on weather conditions at the region from port of Savannah to Fort Stewart every 20 minutes, and detects the update on weather condition at this region using a temporal event detector. Whenever an update event is signaled, the system takes the action of notifying Todd by email and delivering the updated result using a specific web URL pointer.

Note that the action of displaying the updates of weather condition at the specified Savannah region, and the action of reporting to Todd by sending mail is implicitly inferred by the system, based on either the fact that Todd is the owner (creator) of this continual query **Savannah_weather_watch** or the fact that the creator of this continual query has entered a special request that the update results be sent also to his/her manager, Todd, at the CQ installation time.

Interesting to note is that the trigger condition and the query component in a continual query both can be specified in SQL -like expressions. When the trigger condition is defined over the same set of objects as the query component, the FROM clause may be omitted (recall Example 2). Here is an example where the trigger condition is defined over a set of object classes that are different from those over which the query component is defined:

```
Creat CQ Savannah_weather_watch as
Query:
   SELECT plan_no, plan_desc, plan_alt_routes
   FROM   Transportation_plan
   WHERE  plan_route like 'Savannah to Fort Stewart';
Trigger:
   FROM  www.wns.nova.gov
   WHERE location like 'Savannah' AND state = 'Georgia'
```

```
    OR location like 'Fort Stewart';
Stop:  next 3 months.
```

This continual query amounts to saying that "monitoring the weather condition between port of Savannah and Fort Stewart in the next 3 months, provide me with a list of alternative plans whenever the weather condition changes in the region between Port of Savannah and Fort Stewart Reservation". Note also that the **Transportation_plan** may be stored in a relational DBMS (e.g., Oracle), a structured data source, and the weather information is available from the NWS website, a semi-structured data source.

Another interesting feature of the CQ system is to allow users to specify their trigger conditions using system built-in functions in addition to the common string comparison operators such as **CONTAINS**, **LIKE**, and arithmetic operators $<, \leq, >, \geq, =, <>$. For example, the system built-in functions for trigger specification include increased by x percent, denoted as **IncreaseBy%**$(X) \leq x$, and decreased by y percent, denoted by **DecreaseBy%**$(Y) \leq y$, where X and Y are field names of the source data items. Using these system built-in functions, the continual query, "notify me in the next two weeks whenever the stock price of Bayer drops by 5%", can be expressed conveniently as follows:

```
Creat CQ Bayer_Stock_watch as
Query:   SELECT company_symbol, stock_price, hi_last_wk, lo_last_wk
         FROM   Stock
         WHERE  company_name = 'Bayer AG';
Trigger: Stock.company_name = 'Bayer AG' AND
         Stock.stock_price DecreaseBy% 5;
Stop:  9:00:00 am, Oct. 26, 1998
```

Generally speaking, in specifying a continual query, the **Query** clause, **Trigger** condition clause, and **Stop** condition clause are essential and thus mandatory. When there is nothing entered for the **Stop** condition, a default value (e.g., two weeks) is used. When nothing is filled in the **Query** clause, an error message is generated. When the trigger condition is not specified explicitly, the default is set to a time-based trigger at a default time interval (say everyday). In addition, one can specify other optional properties for a continual query, such as timing constraints, contingency plans, and external events. Timing constraints include deadlines, priorities/urgencies or value functions. Contingency plans describe alternative actions to be executed in case the timing constraints cannot be met.

Chapter 3

System Architecture

The *Continual Query* (CQ) system allows users to define their continual queries over multiple remote and possibly heterogeneous data sources. Once a continual query is installed with the CQ system, the CQ server will treat it as a persistent object whose life cycle begins at the installation time and ends whenever its stop condition is met. For each installed continual query, the CQ server will send a notification alert to its owner whenever the updates at the data sources satisfy the given update thresholds (e.g., every 3 days or when the price of IBM stock drops). The CQ server will trigger the execution of the query and return the result to the user.

The CQ system employs the Mediator/Wrapper architecture to provide uniform access to multiple and heterogeneous data sources. We incorporate the distributed query scheduling facilities [32] of DIOM [26], a mediator and wrapper based information mediation system for distributed and interoperable information integration and management, in the design and implementation of the CQ system.

In the following sections, we will briefly describe the concept of mediator and wrapper systems, the CQ system architecture, and the DIOM distributed query processing components. Although the DIOM interoperable architecture and its adaptive query mediation framework has been extensively covered in [22, 26], to make this thesis self-contained, in this chapter we present a brief overview of the fundamental points of the DIOM project. Our attention is more concentrated on the parts of the DIOM previous research that are directly related to the investigation of the continual query processing and optimization.

3.1 Overview of the Mediator/Wrapper Concept

We view an open distributed information system (World Wide Web) as a dynamic interconnection between information consumers (e.g., normal web surfers) and information producers or sources (e.g., an online bookstore), instead of just as a static data repository system. In such a system, it has always been a challenge for various information consumers to query and monitor information and their updates from multiple and disparate information producers. Two issues that arise immediately

are: (1) heterogeneity of information producers' data sources(text files, relational databases, object-oriented databases, or bibliographic databases, etc.) and query capabilities as well as information consumers' query requests, and (2) scalability of distributed query services in the presence of a growing number of information sources [1] and the evolving requirements of both information sources and information consumers.

In order to provide uniform access to information sources and to support more scalable and seamless information integration, the notion of Mediator/Wrapper was introduced [39]. A wrapper is a software component that transforms data or queries from one model to another. More concretely, a wrapper is always tied to a particular data source and a particular information mediator or broker. It provides the given mediator with some customized access to the particular information sources. Thus a wrapper is data source-specific. In multi-agent systems [8, 9] a wrapper is called a resource agent. In contrast to the wrapper concept, a mediator is a software component which represents an information consumer's view of data with respect to a particular domain. A mediator employs wrappers to bridge the gap between the application domain and the information sources. Therefore, a mediator is domain-specific. In the Mediator/Wrapper architecture, both the mediators and the wrappers are extensible as the requirements of applications or information sources change. Figure 3-1 shows a networked architecture of mediators and wrappers.

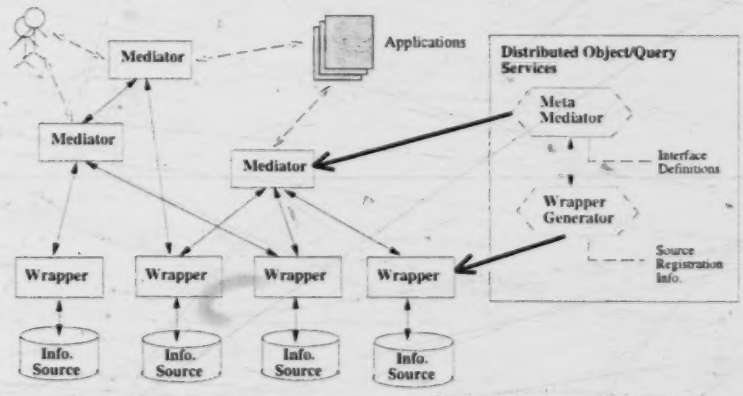


Figure 3-1: The cooperation network of mediators and wrappers

The network of mediators and wrappers allows a seamless incorporation of new information sources into the CQ system. The clean separation of wrapper and mediator functionality allows the distributed query services to be developed as source-independent middleware services which establish the interconnection between consumers and a variety of information producers' sources at the query processing time. As a result, the addition of any new sources into the system only requires each new source to have a CQ wrapper installed. The DIOM services can dynamically capture the newly

[1] We may use both information source and data source interchangeably in the thesis for narrative convenience. However, they hold identical semantics.

available information sources and incorporate them into the distributed query scheduling process.

The information sources at the bottom of the diagram in Figure 3-1 may be one of the following types of sources:

- *well structured*: such as relational or object-oriented database management systems,
- *semi-structured*: such as HTML files, bibliographical record files, other text-based records, or
- *nonstructured*: such as technical papers or reports, program source files, a collection of raw image files, etc.

Each information source is autonomous – it may make changes without approval from the mediators. If, however, an information source makes a change in its export schema, including logical structure, naming, or semantic constraints, then either it notifies the CQ/DIOM object server, or the CQ/DIOM server will send out its robots periodically to check out if the data source has changed its content structure.

3.2 Overview of the CQ System

The goal of the CQ system is to develop a toolkit for update monitoring with event-driven delivery in an open and dynamic evolving environment such as the Internet and intranets. We pursue this goal along two dimensions: The first dimension is to develop a set of methods and techniques that can incorporate distributed event-driven triggers into the query evaluation and search process to enhance information quality and improve system scalability and query responsiveness. The second dimension is to build a working system that demonstrate our ideas, concepts, and techniques developed for continual queries using real-world application scenarios. The method and key techniques of the CQ system development include:

- using the notion of continual queries to support customized (or personalized) update monitoring based on users' preference and requirement (user pull followed by server push),
- incorporating pure push with pure pull data delivery mechanisms in the continual queries service provision,
- integrating distributed query processing and dynamic optimization techniques into the continual query evaluation process for achieving effectiveness and responsiveness of the system.

The first generation of the CQ system has a three-tier architecture: client, server, and wrapper/adapter, as shown in Figure 3-2. This architecture is motivated by the need for providing efficient support to composite event detection and complex condition monitoring of installed continual queries, and the need for sharing information among structured, semi-structured, and unstructured remote data sources.

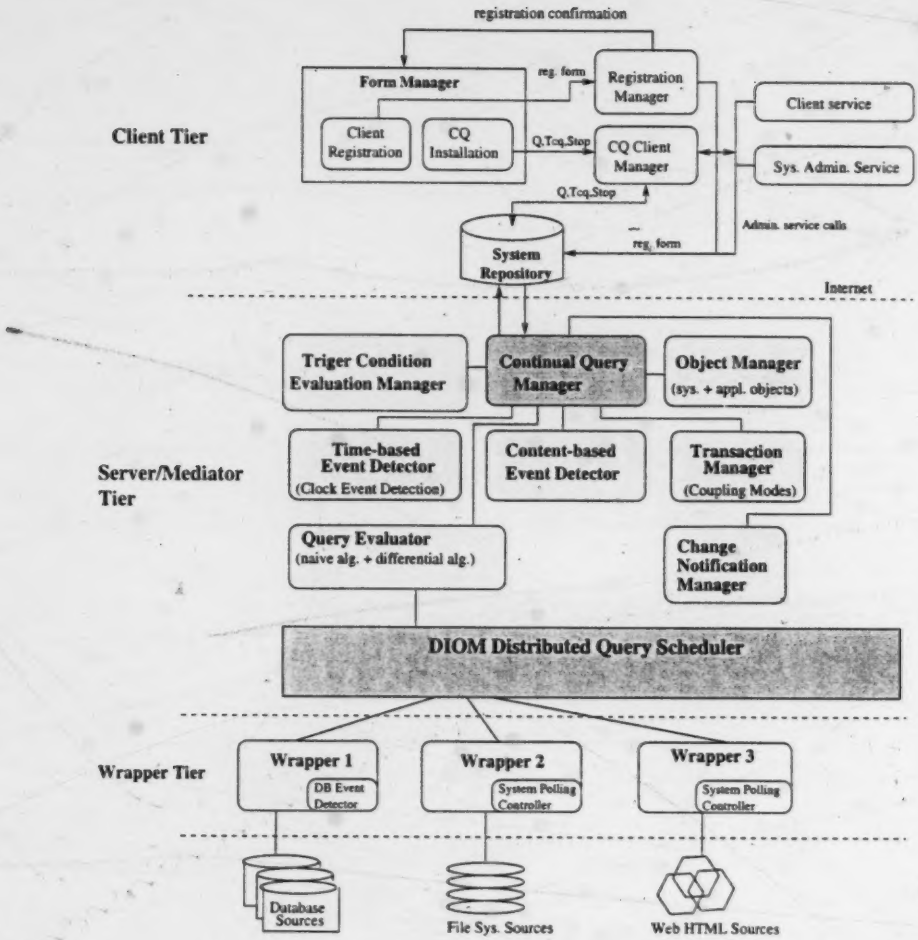


Figure 3-2: A Sketch of the CQ System architecture

The client tier is primarily responsible for receiving users' request and expressing such request in the form of CQ query Q_{cq}, CQ trigger T_{cq}, and CQ termination condition `Stop`. The client manager is also in charge of user registration and providing CQ users with system utilities such as browsing or editing installed continual queries. The **client tier** currently has four components:

- Form manager

  Provides the CQ clients with fill-in forms to register with the CQ system and install their continual queries;

- Registration manager

  For clients who are the first time users of the CQ system, they have to register to CQ system to obtain their userid and password as well as to specify their notification means and email

addresses. The registration manager will record the user information (user id, names, title, password, email address, subscription duration) into CQ system. Upon success, the registration manager will return the client a confirmation by a Web page and an email on his registration.

- Client and system administration services

  Provide utilities for browsing, deleting, or updating installed continual queries, for testing time-based triggers and content-based triggers, and for tracing the performance of update monitoring of source data.

- Client manager

  Coordinates different client requests and invokes different external devices. For instance, once a continual query request is issued, the client manager will parse the form request and construct the three key components of a continual query (Q, T_{cq}, `Stop`), before storing it in the CQ system repository.

  Although not a direct part of the CQ system, one could imagine value-added update monitoring services based on CQ, where a continual query request can be posted in natural language through either voice or hand-writing or both. Recall the example given earlier: "*notify me whenever IBM stock price rises by 5%*". By hooking up the CQ client with a natural language text recognizer, we can parse this request and automatically generate the query, the CQ trigger, and the `Stop` condition for this request. The results can be returned to the user also by multiple methods, such as by email, by fax, or bulletin posting, or by displaying signals on users' desktop screens. User can choose to have the whole result of the CQ delivered to his desktop or just get a brief electronic notification, such as an email or a single beep message on the desktop.

The second tier is the **CQ server** which consists of the following components:

- Continual query manager (CQM)

  The CQM is the top level server component which coordinates with event detectors and condition evaluator as well as other server components.

- Event detectors (ED)

  There are two types of event detectors: *time event detector (TED)* and *content-based event detector (CED)*. TED is to capture time-related events, such as "at 10:00am every Monday" or "every 10 minutes". CED is capable of detecting the updates at the information source which satisfy the update threshold specified in the content-based trigger condition, e.g. "*stock price of Microsoft has dropped by 5%*" or "*total amount of orders received is more than half amount of the products in stock*".

- Trigger condition evaluation manager (TCEM)

  The TCEM is in charge of evaluating the trigger condition for each installed continual query whenever the time events or the content update events of interest are detected and signaled by the event detectors.

- Query evaluator (QE)

  The QE evaluates each user-installed continual query whenever its trigger condition T_{cq} is evaluated to be true. It also provides a guard for the `Stop` condition to guarantee the semantic consistency of the continual query (Q, T_{cq}, `Stop`). The current implementation of the QE component uses two alternative algorithms: one is called *Naive Algorithm*, and the other is called *Differential Evaluation Algorithm*. We will discuss these algorithms in Chapter 5. Each of these algorithms is suitable for certain situations or certain application domains. It allows the system to apply different algorithms for different user queries. The query evaluator is hooked up with the DIOM Distributed Query Scheduler (see Section 3.3) which enables the CQ system to scale up in order to handle hundreds of different information sources.

- Object manager (OM)

  The OM manages all the objects within the CQ system, including system objects such as user registration objects (recording user information and identified by *userid*), continual query objects (each is identified by a *cqid* and recording information about the given continual query, including a query component, a trigger condition, a stop condition, and the installation timestamp), cache for each continual query execution; and application objects such as the objects that the client wants to monitor over the information sources.

- Change notification manager (CNM)

  Once a continual query has detected the updates of interest and fired a new round of execution for the user query, the CQ system must prepare to inform the user about the new result. The CNM is responsible for obtaining the result from the query evaluator and construct an email notification to the user. The notification could be just a URL which tells the user where the new result can be fetched or the whole result of the query if the user wishes to do so.

- Transaction manager (TM)

  When a user-installed continual is executed, a series of programs are invoked. The CQ system must guarantee the execution of the event detector, trigger condition evaluation manager and change notification manager to be predictable, following the designated execution synchronization sequence, and under the control of the system. Just like the transaction management in traditional database systems, we introduce the transaction manager in CQ system. The responsibility of TM is to guarantee atomicity, serializability, and durability of all the processes (we call them transactions).

The third tier is the **CQ wrappers/adapters tier**. The CQ query evaluator and the event-driven update monitor talk to each information source through a CQ wrapper. A information source could be structured (e.g., a relational database) or semi-structured (e.g., an HTML or XML page, a bibliography file), or even non-structured(e.g., a text Perl script). A wrapper is needed for each source because each one has a different way of requesting data and a different format for representing its results. Each wrapper is a specialized data converter that translates the query into the format understood by the remote data source. As the result comes back, the wrapper packages (translates) the response from the source into the relational database format used by the CQ system. Interesting to note is that a CQ wrapper to a data source S is a specialized version of a wrapper to S in the sense that a CQ wrapper, in addition to the responsibilities of a normal wrapper, needs to provide the difference function that can compare the current result obtained from the source with the previous result and tells the CQ server if there is any change to the source data of interest.

3.3 Information Integration in CQ - DIOM

For global data sharing and access over heterogeneous information sources in a distributed environment, the CQ system utilizes the mediator approach and incorporates a mediator architecture called DIOM [26, 18] prototyped in the Department of Computing Science, University of Alberta. The Distributed Interoperable Object Model (DIOM) introduces the approach that explicitly defines the interfaces of an information consumer and connect information consumer's requests with available information producers dynamically. Such run-time interconnection allows the DIOM system to be able to interoperate and scale up with growing number of autonomous and heterogeneous data sources as components. Figure 3-3 shows a sketch of the DIOM prototype system architecture. DIOM keeps user profiles to gather the knowledge on user query requests and capture the query objects in DIOM interface definition language. DIOM wrappers are utilized to bridge the gap between the interoperable database systems and the individual component repositories.

In what follows, we will briefly review the distributed query mediation service provider component. Readers who are interested in discussion on the other components of the DIOM system architecture may refer to [17].

The main task of a distributed query mediation service provider is to coordinate the communication and distribution of the processing of information consumer's query requests among the root mediator and its component mediators or wrappers (recall Figure 3-1). [26] has proposed the general procedure of a distributed query scheduling process in DIOM. It primarily consists of the following steps to process a user query submitted to the DIOM server:

1. query routing,
2. query decomposition,
3. parallel access plan generation,

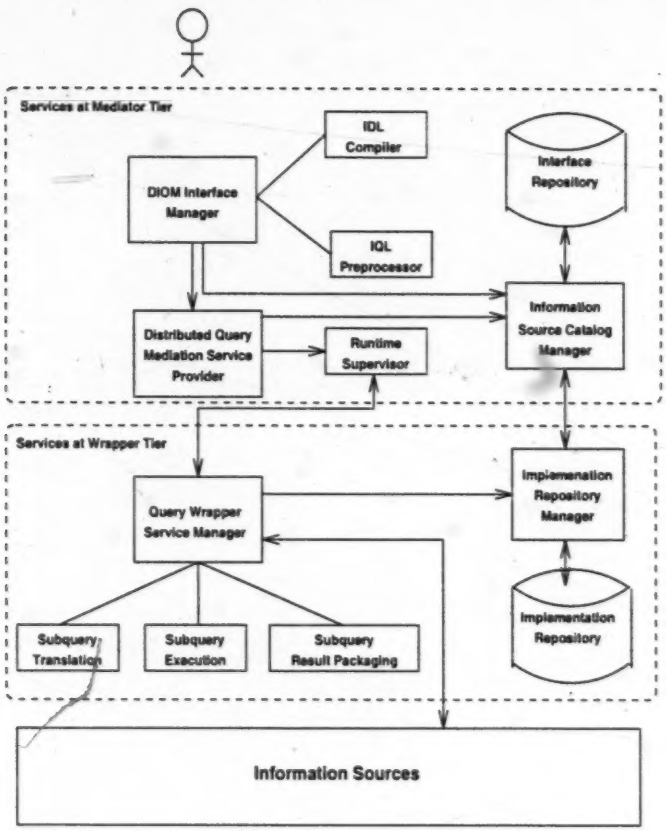


Figure 3-3: DIOM System Architecture

4. subquery translation and execution, and

5. query result assembly.

Given a continual query defined over multiple heterogeneous data sources, its query component and trigger condition evaluation component will be processed using the CQ query processor which is built on top of the DIOM distributed query scheduler. Figure 3-4 shows the steps of how a user query from the CQ system is processed inside the DIOM distributed query scheduler.

Query routing is the first step. The main task of query routing is to select relevant information sources from available ones for answering the query. This is done by mapping the domain model terminology to the source model terminology, by eliminating null queries, which return empty results, and by transforming ambiguous queries into semantic-clean queries. Consumers' query profiles and producers' data source profiles play an important role in establishing the interconnection between a consumer's query request and the relevant information sources.

The second step is called *Query decomposition*. It is done by decomposing a user query expressed

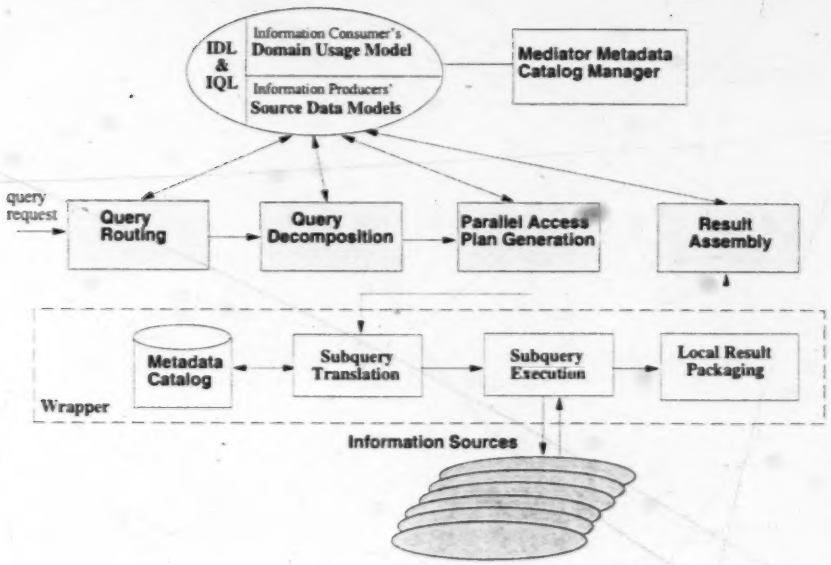


Figure 3-4: A Sketch of the Distributed query scheduling framework in DIOM

in terms of the DIOM interface query language (IDL) into a group of subqueries, each targeted at a single data source.

The third step is *Parallel query planning* where optimization of distributed query takes place. The goal of generating a parallel access plan for a group of subqueries is to find a relatively optimal schedule that makes use of the parallel processing potentials and the useful execution dependencies between subqueries, resulting from built-in heuristics, to minimize the overall response time and reduce the total query processing cost.

The next step is called *Subquery translation and execution.* The translation process basically converts each subquery expressed in the interface query language into the corresponding information producer's query language expression, and adds the necessary join conditions required by the information source system.

After submitting the subqueries, the DIOM query scheduler is responsible for (1) packaging each individual subquery result into a DIOM object (done at wrapper level) and (2) assembling results of the subqueries in terms of the consumers' original query statement (done at mediator level). The semantic attachment operations and the consumers' query profiles are the main techniques that we use for resolving semantics heterogeneity implied in the query results. This step is referred to as *Query result packaging and assembly.*

3.4 Client-Server Design

Depending on the need of the application, the CQ client manager, the CQ trigger evaluator, the event-driven update monitor, the query router, query planner, and query result assembler could be located on a single machine, or distributed among several computers connected through local or wide area networks. The CQ system uses the most flexible client-server arrangement which is customizable with respect to the particular system requirement of the applications. Figure 3-5 shows three different scenarios for multi-layer client/server coordination among the CQ components.

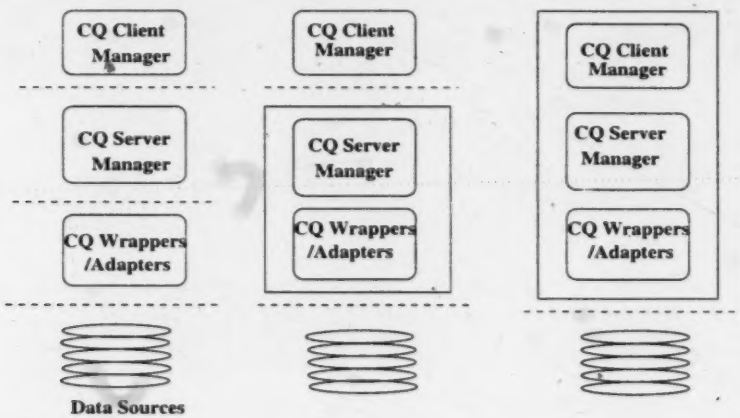


Figure 3-5: Example Scenarios of the Client/Server coordination among CQ components

In this chapter, we have briefly introduced the main components in CQ systems and presented the general picture of the main architecture and the application domain. The rest of the chapters in this thesis will present the continual query execution model, the differential reevaluation algorithms for efficient processing of continual queries, and the design and implementation details of the first CQ prototype system in the context of this general picture.

Chapter 4

Continual Query Execution Model

The CQ system presents an extensible architecture for experimentation with push-enabled data management and event-driven data delivery systems. Rather than starting from scratch, the CQ system takes advantages of the conventional pull-based DBMS services to the extent possible and adds push-based data delivery elements to them when it is necessary. For instance, rather than introducing a new event-driven query language, the CQ system uses continual queries to express trigger conditions and unifies them with the regular SQL queries so that continual queries may be expressed in SQL with minimal additions to the language. In particular, the execution model of the CQ system provides different communication protocols between triggering transactions (update transactions) and triggered queries and alert messages (such as sending email notification, firing a subsequent execution of a query). These communication protocols range from synchronous or asynchronous cooperation, causally dependent or independent scheduling, deferred or immediate notification, to execution of triggered actions in a same or separate transaction as the triggering event.

In Chapter 2 we have presented the CQ extension to the conventional DBMSs and pure pull web search systems and overviewed the continual query specification language, which is a slight extension of SQL with the primitives introduced in Chapter 2. We illustrate the continual query specification, in particular CQ trigger definitions, through a number of examples. In Chapter 3 we have described the CQ system architecture. In this chapter, we discuss the execution model of continual queries. We will describe the optimization techniques for continual query processing in Chapter 5 and report our implementation design of the first CQ prototype system in Chapter 6.

4.1 A Quick Look at Continual Query Execution

Let us first take a quick look at the the execution process of a continual query. For more details on the execution model of continual queries, see Section 4.2.

Recall the continual semantics described in Chapter 2, it specifies that, for each continual query CQ_i, denoted by $(Q, T_{cq}, \texttt{Stop})$, the first execution of CQ_i is activated by the installation of the CQ_i,

without going through the evaluation process of the condition $T_{cq} \wedge \neg$**Stop**; whereas the subsequent executions of this continual query are fired only when the condition $T_{cq} \wedge \neg$**Stop** becomes true. More concretely, when a continual query CQ_i is entered (installed) the first time, the following activation actions take place:

- CQ_i is registered with a unique continual query identifier (cqid);
- Rather than activated by the condition evaluation manager, the first run of CQ_i is fired by the continual query activation manager (see Section 4.2.2 for further detail), which executes the query component Q. There is no verification on trigger condition or stop condition except simple syntax check. The first run of CQ_i will return the whole answer of Q, and cached the answer as the previous execution result of CQ_i;
- The trigger condition T_{cq} is activated in the sense that the update events of interest are identified, each associated with a conditional event; and the trigger activation variables (such as transaction coupling mode, dependency coupling mode, schedule coupling mode, and execution coupling mode) are initialized.

The subsequent runs of CQ_i will be fired whenever the trigger condition T_{cq} is evaluated to be true, and the termination condition **Stop** is not true. Each subsequent execution of CQ_i proceeds as follows:

- Step 1: *Update Events Identification*
  This step is to identify the update events of interest from the trigger condition expression of CQ_i. It is done by decomposing the trigger condition T_{cq} into a list of T_{cq} triplets, each triplet consists of a basic update event, an atomic conditional event, and a connector to the next triplet in the list;
- Step 2: *Update Events Detection*
  This step is to decide when to detect the changes and what to detect for the given trigger condition, and which event detectors should be used. For each triplet generated in the Step 1, the atomic condition is evaluated when the basic update event is *signaled*;
- Step 3: *Logical Events (Condition) Evaluation*
  This step is carried out by the condition evaluation manager, which first select a triplet from the list of T_{cq} triplets generated in Step 1, if the connector is an AND connector (or an OR connector), the AND logical event detector (the OR event detector) is invoked; if the connector is WHERE, the next triplet in the list will be used as an add-on condition to the basic update event component of this triplet; and so on. For further detail, see Section 4.2.3;
- Step 4: *Differential Query Execution and Result Delivery*
  If the condition evaluation in step 3 returns a *true* value, then the following pre-defined actions

are scheduled to execute: (1) fire the next execution of the query component Q, (2) compute the difference between the current run of Q and the result of the previous run, (3) notify the user of the arrival of new updates of interest, and (4) deliver the differential result to the user.

A walkthrough example to illustrate this process is provided in Section 4.2.

4.2 Continual Query Execution Model

We have explained how one defines continual queries in the previous section. We now describe the implementation of how the CQ system triggers and executes continual queries.

It is well known that in a conventional pull-based DBMS user application programs are executed when explicitly requested to do so. Execution of such programs typically results in the processing of a sequence of *transactions*, where each transaction is a unit of consistency and recovery. The system guarantees *atomicity* (all updates issued by the transaction are installed in the database or none are), *serializability* (the concurrent interleaved execution of a set of transactions is equivalent to a serial no-interleaved execution), and *durability* (once a transaction is committed, its updates will never be rolled back). In contrast, a continual query system must evaluate installed continual queries under system control (not user or application control). More concretely, once a continual query is installed, the system must decide not only how to detect the update events of interest, how to evaluate the trigger condition, and when to fire the subsequent execution of the query component, but also how the execution of these tasks should be treated with respect to user transactions. The continual query execution model is an attempt to answer these questions.

4.2.1 Basic Coupling Modes

Continual queries in practice are often defined over multiple, autonomous and possibly heterogeneous data sources. The local update transactions are usually orthogonal to the continual queries specified over the same set (or a subset) of data. Furthermore, both trigger condition evaluation component and query component of a continual query are side-effect free transactions. Due to the autonomy and distribution of data sources and the side-effect free nature of continual queries, it is not only important but also practical to allow a more flexible execution model.

A flexible execution model allows trigger condition evaluation and query execution to be broken off into different execution threads from the triggering transaction (the transaction that carried out the update operations). More concretely, it should be possible to allow the continual query evaluation to be separated from the (triggering) transaction that carried out the actual updates. This would allow the triggering transaction to commit earlier, and would potentially increase concurrency and reduce wasted work (rollback of incomplete transactions after a crash). The CQ system execution model for continual queries uses the notion of coupling modes to provide this flexibility.

In the CQ system, we support four basic coupling modes: transaction coupling mode: *separate* or *same*, execution coupling mode: *asynchronous* or *synchronous*, dependency coupling mode: *causally*

dependent or *causally independent*, and schedule coupling mode: *immediate* or *deferred*. We view the execution model of each continual query to consist of the following four participating transactions:

- (1) the triggering transaction that carries out the update operations,
- (2) the update event detection transaction that detects if the data of interest has been updated,
- (3) the trigger condition evaluation transaction that evaluates the condition based on the newly updated data, and
- (4) the transaction that carries out the subsequent execution of the query component and sends out the alerts or change notification messages.

Such arrangement provides more flexibility for utilizing multiple execution threads and parallel execution for continual query processing, which are critical techniques to the effectiveness and responsiveness of a push-enabled distributed cooperative information management system.

In the CQ system, it is possible that the coupling case for transaction types (1) and (2) may be different from the coupling case for transaction types (2) and (3) as well as the coupling case for transaction types (3) and (4).

We illustrate the meanings of each coupling mode using the coupling scenario for transaction types (2) and (3), which relates to the trigger condition part of the continual queries. For the trigger condition part of a continual query, the coupling mode specifies when the condition is to be evaluated relative to the triggering event (i.e., the update event being monitored):

- **Transaction coupling mode**: separate or same

  The transaction coupling mode *separate* means that the condition evaluation triggered by the update event runs as a separate transaction with respect to the transaction that detects the update events of interest.

  The transaction coupling mode *same* means that the condition evaluation triggered by the update event runs either as part of the transaction for detecting the update event in the case that the updates performed by the triggering transaction are local operations, or as part of the triggering transaction in the case that the updates are performed by the same user or application program who installed the continual query.

- **Execution coupling mode**: asynchronous or synchronous

  The asynchronous coupling mode means that the update event detection transaction may run in parallel with the trigger condition evaluation transaction.

  The *synchronous* coupling model means that if the trigger condition evaluation transaction is triggered by the transaction that detected the update events, then the trigger condition evaluation transaction is executed, and the execution control returns to the 'triggering' transaction only after the condition evaluation transaction is committed.

- **Dependency Coupling Mode**: casually dependent or casually independent

  The *casually dependent* coupling mode means that the trigger condition evaluation transaction can be scheduled only after the 'triggering' transaction that detected the update event has committed.

  The *casually independent* coupling mode means that the scheduler is free to schedule the trigger condition evaluation transaction independently of the update event detection transaction when the update transaction is local.

- **Schedule Coupling Mode**: immediate or deferred

  The schedule coupling mode *immediate* means that the trigger condition evaluation transaction is fired as soon as the triggering transaction commits. When the updates are carried out by a global update transaction issued by the same user or application program, the triggering transaction refers to this global update transaction. When the updates are carried out by local transactions or other remote and autonomous transactions, the triggering transaction refers to the update event detection transaction.

  The schedule coupling mode *deferred* means that the CQ trigger condition evaluation is fired at the end of the update event detection transaction and before it commits.

  By looking into the semantics implication of these coupling modes, We come to the following conclusion: The schedule coupling mode *deferred* must be used in conjunction with the *same* transaction coupling mode.

The *same* transaction coupling mode can be used only in conjunction with synchronous execution coupling. The *deferred* schedule mode is applicable only in conjunction with the *same* transaction coupling mode. However, the immediate schedule mode can be used in conjunction with both *same* and *separate* transaction couplings. Also both dependency couplings are applicable only to *separate* transaction coupling, *immediate* schedule coupling, and *asynchronous* execution coupling.

In a similar manner, we may illustrate the possible coupling cases for transaction types (1) and (2), the event detection part of the CQ, and for transaction types (3) and (4), the query scheduling part of the CQ. For the query scheduling part of a CQ, each coupling case specifies when the subsequent run of the query component is to be fired relative to the trigger condition evaluation transaction.

In the CQ system we allow users to define their application-specific coupling modes for any of the three pairs of the participating transaction types. In the absence of user-specified coupling modes, the system default coupling case will be used. The default coupleing modes are:

- **Between (1) and (2): separate, asynchronous, causully independent**
- **Between (3) and (4): separate, synchronous, causully dependent**

4.2.2 Continual Query Installation

Once a continual query CQ_i, denoted by $(Q, T_{cq}, \texttt{Stop})$, is defined, the user may install it directly to the CQ system. At the installation time, the **Install** module of the client manager takes the continual query and passes it to the CQ server. The server activates it using the **activate** command. The activation process consists of the following three main tasks:

- making this continual query a persistent object and generating a unique identifier (**cqid**) for it;
- execute Q for the first run of CQ_i and cache the answer as the previous run result of the query component;
- Initializing the execution attributes and data structures used for event detection and condition evaluation of this given CQ. This task includes decomposing the user-specified CQ trigger condition into a set of triplets, each triplet is described by a basic update event, an atomic conditional event, and a connector; and setting up the initialization for the transaction coupling mode, the dependency coupling mode, the schedule coupling mode, and the execution coupling mode (recall Section 4.2.1).

The **Activate** command also returns a handle that will be used to deactivate this continual query when its termination condition is expired.

Users can use the **activate** command to define the coupling modes according to application specific requirements. The syntax of the **activate** command is given below:

```
Activate <cqid>
  define communication protocol between
      <trans1> and <trans2>
  TransactionCoupling = same | separate
  ExecutionCoupling = synchronous | asynchronous
  DependencyCoupling = causally dependent | causally independent
  ScheduleCoupling = immediate | deferred
```

Once a continual query is activated, it runs continually following the communication protocol defined by the specific coupling case. The continual query is terminated when its **Stop** condition is evaluated to be true. To terminate an installed continual query, the command **Deactivate <cqid>** is invoked, which removes from the CQ system catalog the corresponding continual query object identified by **cqid**, deactivates the related event detectors that are still active, and sends to the owner of this CQ a notification that this CQ is expired.

4.2.3 Event Detection

The main task of event detection manager is to decide what to detect, when to detect, and how to detect. The decision is made based on the update events identified from the trigger condition specification and the type of events to be detected. As discussed in Section 2.4.1, the trigger condition part of a continual query may be a primitive event, such as a *temporal event*: every two days or every first day of the month; an *atomic conditional event*: the stock price is greater than 100 (`price > 100`); or a *composite event*, which is formed by an event composition expression of the form "E_1 `<event_op>` E_2", where E_1 and E_2 are primitive or composite events. Typical examples of composite events are

```
Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5
keyword CONTAINS 'Java' OR keyword CONTAINS 'JDBC'
qty_on_hand(item) > threshold(item)
qty_on_hand(item) + qty_on_order > threshold(item)
```

Each primitive event is detected by using a primitive event detector, which is either a basic temporal event detector or an atomic conditional event detector. An operation *signal* is defined for the event entity type, and is executed by the event detector components of the system.

4.2.3.1 Time-based Event Detection

For time-based continual queries, a temporal event detector, or so-called time-based event detector, is used, which translates the time-based trigger condition into a clock event and installs the clock event script to the CQ clock daemon. Whenever the clock event occurs, the trigger condition is signaled. Thus the subsequent execution of the query component is fired. A distinct feature of time-based continual queries is the use of *user-controlled polling* for update monitoring.

There are two key implementation techniques useful for time-based event detection: The first technique is to design a generic transformation program that takes the user-defined time condition and transforms it into a clock event expressed in the clock event scripting language; the clock manager (daemon) will then take over the control and trigger the update event detection according to the clock event installed; whenever the update event is signaled, the continual query manager will call the query evaluator to fire the subsequent run of the query component, and call the change notification manager to deliver the change notification message as well as the update result. The second technique is to develop a clock event manager which, on one hand, provides a scripting language to allow users to specify an arbitrary clock event and the action to be taken if the clock event occurs, and on the other hand, provides triggering capability so that it can fire the specified action (e.g., invoke a program) when a specific clock event is signaled.

4.2.3.2 Content-based Event Detection

In contrast to time-based continual queries, the content-based continual queries use the *system-controlled polling* for update monitoring. Thus, there are more than one strategies possible for implementation of the CQ trigger condition monitoring and event detection.

In order to carry out the content-based event detection, the first thing we need to do is to identify what update events are of interest to the given continual query. As mentioned in the continual query activation procedure (recall Section 4.2.2), for each installed continual query (Q, T_{cq}, `Stop`), its trigger condition T_{cq} is decomposed into a list of T_{cq} triplets, each triplet is described by a basic update event, an atomic conditional event, and a connector. For example, if the trigger condition is "`Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5`", then the following triplets are generated:

```
(Stock.price, Stock.price IncreaseBy% 5, WHERE)
(Stock.company, Stock.company = IBM, OR)
(Stock.price, Stock.price IncreaseBy% 5, WHERE)
(Stock.company, Stock.company = Intel, END)
```

For the trigger condition: `qty_on_hand(item) > threshold(item)`, two triplets are generated. They are: `(qty_on_hand, true, >)` and `(threshold, true, END)`. Note that the connector `WHERE` means that the next triplet is not an update event of interest but a constraint on the current update event. In this case, UPDATE on the stock price is the event we would like to monitor, and the condition `Stock.company = IBM` is simply a constraint, saying that we are only interested in monitoring UPDATE on the stock price of IBM but not other companies' stock prices.

Now we can determine **what to detect** based on the basic update events identified by the list of T_{cq} triplets.

Example 3 Given the trigger condition:

"`Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5`",

the basic events of interests are UPDATE operations on `Stock.price` and `Stock.company`, as well as INSERT and DELETE operations on the object class `Stock`. For trigger condition `keyword CONTAINS 'Java' OR keyword CONTAINS 'JDBC'`, if the condition field name `keyword` is mapped to `Document.title` and `Document.abstract` available at the corresponding data source(s), then the basic events of interests are INSERT and DELETE operations on `Documents` objects, and UPDATE operations on `Document.title` and `Document.abstract`.

The next question is **how to detect**, namely we need to decide which mechanisms may be used to detect the changes made by the update operations, possibly from some transactions that are local to the data source; In the CQ system, we distinguish between the data sources that have built-in

trigger capability such as the data sources managed by trigger-enabled RDBMSs (incl. Oracle, DB2, Informix, Sybase) and the data sources that have no built-in trigger capability such as most of the web sites and file systems.

- For the data sources with built-in trigger facility, the CQ system may install the database triggers on the data columns or objects of interest. Whenever there is an update, the database transaction that carries out this upate will send an update signal to the corresponding CQ wrapper. We provide the host-specific trigger installation program (such as Oracle trigger installation program) to install triggers on those data objects and data columns that are accessible to the CQ system.

- For the data sources with no built-in trigger facility, we use system-controlled polling with system-defined interval (such as every 30 seconds).

Note that the capabilities of database trigger supported in commercial DBMSs today are not sufficient, particularly in those cases where run-time installation of customized database triggers is required, In these situations, a system-controlled polling will be used in conjunction with the database triggers. Our experience tells that not all the RDBMSs allow database triggers to be installed by a remote program through JDBC. In the first prototype of the CQ system, we implement the content-based event detection using the system-controlled periodic polling.

Now, let us walk through the event detection process. Given a continual query CQ_i defined by $(Q, T_{cq}, \texttt{Stop})$. Suppose that the trigger condition T_{cq} has been transformed into a list of T_{cq} triplets, denoted by TripleSet($\texttt{cqid}, T_{cq}$). To simplify the steps (that) we need to walk through, let us assume that the connectors we use in this walkthrough are the most commonly used ones, namely **WHERE, AND, OR, END**. For each triplet in TripletSet($\texttt{cqid}, T_{cq}$), we form a event detection query, denoted by Q_{detect}, which is to be submitted to the relevant data sources to detect if an update is occurred.

- For a triplet of the form $(T.A, T.A\vartheta v, \texttt{AND})$ or $(T.A, T.A\vartheta v, \texttt{OR})$ or $(T.A, T.A\vartheta v, \texttt{END})$, where T denotes the object class, A, B are instance variables of T, and ϑ is the comparison operator, let **prev** denote the value of instance variable A contained in the result of previous execution of the given CQ. Thus, the corresponding event detection query Q_{detect} is expressed as **SELECT A FROM T where A** $\neq$ **prev**.

- For a triplet of the form $(T.A, T.A\vartheta w, \texttt{WHERE})$, we fetch the next triplet, say $(S.B, S.B\vartheta w, \texttt{END})$ from the remaining list of TripletSet($\texttt{cqid}, T_{cq}$). Thus, the event detection query Q_{detect} is expressed as **SELECT T.A, S.B FROM T, S WHERE S.B** ϑ **w AND T.A** $\neq$ **prev**.

4.2.4 Condition Evaluation

In principle, one may want to detect all the update events of interest before starting the trigger condition evaluation process. In practice, the CQ trigger condition evaluation is carried out in

conjunction with the process of basic update event detection, to guarantee the efficiency of the condition evaluation. For example, if a condition is of the form $(T.A\vartheta v_A) \wedge (T.B\vartheta v_B)$, and if the event detection query over the triplet $(T.A, T.A.\vartheta v_A, \texttt{AND})$ returns empty answer, then we can conclude that the trigger condition is false without looking into the second triplet $(T.B, T.B\vartheta v_B, \texttt{END})$.

Now, let us walk through the condition evaluation process for a continual query CQ_i defined by $(Q, T_{cq}, \texttt{Stop})$. Let TripleSet($\texttt{cqid}, T_{cq}$) denotes the list of T_{cq} triplets generated by the CQ activation process. Similar to the discussion on event detection, we simplify the steps we need to walk through by assuming that the connectors used in this walkthrough are **WHERE, AND, OR, END**. The condition evaluation process of CQ_i proceeds as follows:

- Step 1: It starts by selecting a triplet in TripleSet($\texttt{cqid}, T_{cq}$), and then check the connector type of this triplet:
- Step 2: if it is an **END** connector, then this content-based trigger condition is evaluated to be true, and the subsequent query execution is fired.
- Step 3: if it is a **WHERE** connector, let us denote the selected triplet as $(T.A, T.A\vartheta v, \texttt{WHERE})$, and the next triplet is fetched from the remaining list of TripleSet($\texttt{cqid}, T_{cq}$), denoted by $(S.B, S.B\vartheta w, \texttt{AND})$, then the update event detection query Q_{detect} is expressed as **SELECT T.A, S.B FROM T, S WHERE S.B** ϑ **w AND T.A** $\neq$ **prev**. If Q_{detect} returns a non-empty answer, it means the update event has occurred; go to step 6. If Q_{detect} returns an empty answer, we can conclude that the corresponding trigger condition is false.
- Step 4: if it is an **AND** connector, let us denote the selected triplet as $(T.A, T.A\vartheta v, \texttt{AND})$, then the update event detection query Q_{detect} is expressed by **SELECT T.A FROM T where T.A** $\neq$ **prev**. If the answer to this query Q_{detect} is empty, then the condition evaluation is false. Otherwise (i.e., if the answer is non-empty), go to Step 6.
- Step 5: if it is an **OR** connector, let us denote the selected triplet as $(T.A, T.A\vartheta v, \texttt{OR})$, then the update event detection query Q_{detect} is the same as the case for an **AND** connector, i.e., **SELECT T.A FROM T where T.A** $\neq$ **prev**. However, unlike the **AND** connector case, if the answer to this query Q_{detect} is non-empty, then we conclude that the condition evaluation is true. Otherwise (i.e., the answer is empty), we need to go to Step 6.
- Step 6: select another triplet from the remaining list of triplets in TripleSet($\texttt{cqid}, T_{cq}$), and go back to Step 2.

Obviously, the richer set of event composition operators is used, the more sophisticated the event detection process will be. A complete description of event composition operators and their formal semantics is beyond the scope of this paper. Readers may refer to [23] for further details.

4.2.5 Issues on Efficient Condition Evaluation

Users and application programs may define as many continual queries as they wish. Once these continual queries are installed, they run continually as long-running side-effect free transactions with checkpoints [1]. Despite all the query components, each from one installed continual query, the set of all trigger conditions forms a potentially large set of predefined queries (i.e., event detection queries) that have to be evaluated efficiently. Furthermore, the trigger condition component of a continual query may be more sophisticated than the query component when the update monitoring threshold is defined over several different object classes and uses special operators (such as `IncreaseBy%`) that are not supported by the data sources upon which the condition is evaluated. Several techniques have been identified as being useful for performance optimization of the condition evaluation:

The first technique is *Multiple Condition Optimization* and also called multiple query optimization in the literature [33]. This technique represents conditions (and the events that signal the condition evaluation) by condition evaluation graphs, which resemble the query graphs commonly used in query processing. The leave nodes of the graph are triplets of the form $(R, R+, R-)$, where R corresponds to a set of entity instances before the update, $R+$ corresponds to the set of instances inserted into R by the update, and $R-$ the set of instances deleted from R by the update. The internal nodes correspond to operators of some convenient algebra into which the query language can be compiled (e.g., select, project, join). The key idea of multiple condition evaluation consists of identifying common subgraphs, and evaluating these subconditions once for a whole set of queries, instead of once for every query [33]. For a continual query system, the common subconditions may be detected at the algebraic level due to the distribution and autonomy of data sources, whereas in a centralize data base system the common subconditions may also be detected at the lower level (e.g., use common access paths). The multiple query evaluation problem is complicated by the need to ensure that the conditions will have to be evaluated simultaneously; e.g., they are triggered by the same update event.

The second technique is *Incremental Condition Evaluation*. A main task of continual query evaluation is to determine whether the answer to a previous execution of the query component (say at time t) has changed as a result of some update event to some of the query's operands at time t'. Let Q be a query defined over an entity set R, and $Ans(Q,t)$ be the answer to the query Q at time t. Let $R' = (R \ minus \ R- \ union \ R+)$. A brute force method for computing the change in $Q(R,t)$ would be to compute $Ans(Q,t') = Q(R')$, and then the symmetric difference of $Ans(Q,t)$ and $Ans(Q,t')$. Incremental evaluation computes this symmetric difference directly from $R+, R-$, and Q. Sometimes R is also needed when Q involves joins [25]. Many algorithms have been proposed in view materialization research for incremental maintenance of materialized views (see Chapter 7 for reference), and may be directly deployable for incremental condition evaluation in the continual query systems.

[1] Each time when the trigger condition is evaluated to be true and the query is fired is referred to as a checkpoint.

An extreme case of incremental condition evaluation is the situation where it may be possible to infer that there is no change in a query's answer with respect to an update event without evaluating the query. Put differently, we can ignore an update event E at t' with respect to the execution of query Q at t, if we can tell that the symmetric difference between $Ans(Q,t)$ and $Ans(Q,t')$ is empty by looking only at the update event E and query expression Q. A trivial example is the update event that modifies a data object that is irrelevant to the query Q. A less trivial example is an update that modifies the Intel stock price to a higher value; clearly, this update event is ignorable with respect to the trigger condition `stock.price(Intel) DecreaseBy% 5`.

Also more opportunities for optimization may arise out of the interplay between the event detection, the condition evaluation, and the subsequent execution of the query component. Generally speaking, more work is needed to develop heuristics and cost models that the condition monitor can use to explore the tradeoffs and benefits of these tactics and algorithms.

4.3 Performance Evaluation Issues

We have presented a design and selection of alternative architectures and algorithms for a distributed push-enabled data management system that supports continual queries. Research on push-enabled continual query systems must be accompanied by a careful performance evaluation effort. For the CQ system, such an effort is under way. The goal of the first effort is rather modest, that is to verify that a continual query system can indeed outperform a pull-based passive data delivery system for applications that require time-constrained update monitoring. Towards this objective, a simple condition monitor and a small situation monitoring application were implemented using C, Perl, JDK1.1, JDBC, and Oracle 7.0, upgraded to Oracle 8.0. Three types of data sources are used in this prototype: (1) an Oracle database which is remotely accessible through SQL, OraPERL, and SQLNet; and a Microsoft SQL server database which is remotely accessible through JDBC and SQL; (2) a collection of semi-structured UNIX files which are accessible through Java applets and Java servelets. (3) a World Wide Web HTML source which is accessible through our html wrapper and filter utility. We are planning to do a simple experiment, making a comparison between user polling and continually monitoring using continual queries. We expect (with confidence) that this simple experiment will verify the hypothesis that push-enabled data delivery system can outperform (ad-hoc) polling over a pull-based passive data delivery system when the number of objects being updates and monitored is proportionally large.

We are also interested in planning careful controlled experiments for comparing the performance of alternative condition evaluation tactics, as well as effort on studying architectural alternatives for the push-enabled continual query systems and their impact on performance, and possibly building a performance testbed for studying the extent to which the final design of the CQ system is able to meet or exceed the processing requirements of a distributed time-constrained update monitoring and event-driven information delivery system.

Chapter 5

Differential Evaluation Algorithm for Continual Queries

5.1 Motivation

Continual queries are standing queries that run continually until the termination condition becomes true. Whenever an relevant update is performed, the CQ system will trigger the execution of the corresponding continual queries. It is obvious that the subsequent execution of a given continual query is only interested in those data that have been updated since the previous execution. In the situation where the amount of updates is small, one way to optimize the subsequent executions of a given CQ query is to use differential evaluation method such that queries that can be answered using delta information (i.e., the updated data) rather than the full set of base data. Similar techniques have been widely used in incremental view materialization [5, 14, 19, 16, 34].

More concretely, recall Chapter 2, we have defined a CQ query as a sequence of query results, modeled by $Q_{cq}(S_1)$, ..., $Q_{cq}(S_n)$, where $Q_{cq}(S_i), (i = 1, ..., n)$ is the result of running Q_{cq} on the database state S_i. Quite often there are situations where users are more interested in the difference between $Q_{cq}(S_i)$ and $Q_{cq}(S_{i+1})$. This can be accomplished by naively executing the entire query and then filtering out the part of the query result that is the same as the previous result. This simple and straightforward approach can be quite expensive, especially in the Internet environment where query results need to be gathered from multiple source data repositories. An obviously more attractive approach is the differential query evaluation method, which is particularly powerful when $Q_{cq}(S_i)$ is relatively large, and only a small percentage of the result changes from state S_i to state S_{i+1}.

For example, suppose we have a join query $R \bowtie S$ and let us assume for simplicity that R is located at site 1 and S is located at another site, namely site 2. Suppose this query is installed as a continual query that runs every 5 hours. It means that the CQ server will check out the updates at the source R and S every 5 hours. Whenever R or S objects at the sources change, the query $R \bowtie S$ will be reevaluated again. Suppose we have a new object o_i added into the source R at time

t_1 after the initial installation of this CQ at t_0, $t_1 - t_0 < 5h$, and this insertion is the only update at source R and source S up to the time point $t_0 + 5h$. Obviously the change effect can be computed by either executing $R \bowtie S$ again and then computing the difference with the previous result, or by computing the net change effect using $\{o_i\} \bowtie S$. When the cardinality of R is very large, the differential computation of net change effect is much cheaper than the re-evaluation of the original query expression $R \bowtie S$ from scratch.

[25] proposes a differential re-evaluation algorithm for continual queries. The key idea behind this algorithm is to produce $Q(S_{i+1})$ by incrementally updating $Q(S_i)$. More concretely, in contrast to a complete re-evaluation, differential re-evaluation means that after the initial execution of a CQ, the re-evaluation of each subsequent execution of this CQ will be performed by using the differential form of the query, denoted as δQ_{cq}. This way, we avoid reprocessing the entire query from scratch. When the changes are substantially smaller compared with the latest query execution result, this differential update will be more efficient than reprocessing the entire query.

The differential re-evaluation algorithm (DRA) is invoked by the CQ manager based on the epsilon specification associated with the given CQ. We assume that the information available when the DRA is invoked includes:

- the CQ specification (Q_{cq}, $Trig_{cq}$, $Term_{cq}$);
- the contents of each base relation after the last execution of the CQ;
- the differential relations for each of those operand relations that have been changed since the last execution of the CQ;
- the timestamp of the last execution of the CQ;
- the complete set of the result of the CQ produced by the last execution.

In short, the Differential Re-evaluation Algorithm (DRA) is developed for incrementally computing the new query result from processing updates on top of the previous result. [24] proves that the differential re-evaluation algorithm - DRA is *functionally equivalent* to the "recompute the query from scratch" solution, and, in many situations is more efficient.

Although the differential re-evaluation of continual queries has been extensively covered in [25, 24], to make the discussion of our implementation design of the DRA easier to understand, and make this thesis self-contained, in this chapter we will first present the notation and an brief overview of the differential reevaluation algorithm, and then describe the design choices and implementation plan that we have laid out for efficient realization of the DRA in the first prototype of the CQ system.

The structure of this chapter is as follows: Section 5.2 describes the notation and terminology required to explain the DRA algorithm. Section 5.3 describes the strategy to generate $Q(S_n)$ from

$Q(S_{n-1})$ incrementally, thus reducing both processing time and network transmission bandwidth. Section 5.6 discusses the implementation consideration of the DRA algorithm.

5.2 Notations and Terminology

The relational terminology is used in this paper to specify continual queries, to record and manipulate changes by differential relations and associated operators. This notation does not constrain our algorithm and solutions to relational database management systems. In DIOM, information consumers formulate queries with GUI tools, which are then translated into appropriate query languages for backend processing, such as SQL. On the other side, information producers need only to generate the differential relations, which are simple tables of update operations, to communicate with the consumers.

We assume that the reader is familiar with the basic concepts and notation concerning relational database, as described in [28]. In this paper we refer to relational selection, projection, join, outerjoin, union, and difference operators by σ, Π, $\bowtie$, `Outerjoin`[1], $\bigcup$, and $-$ respectively. For presentation convenience, we sometimes use $\Pi(R;X)$ to denote $\Pi_X(R)$, $\sigma(R;F)$ to denote $\sigma_F(R)$, and a SPJ expression to denote a `Select-Project-Join` expression.

We use differential relations to represent the net effect of a collection of updates to a relation, either stored or derived. The differential relation for a stored relation is instantiated by the system when the source is updated by insertion, deletion or modification (see Example 4). We define an operator that computes differential relations for arbitrary SPJ expressions. The concept of differential relations is, to some extent, similar to the concept of hypothetical relations used for incremental updating materialized views[5, 14]. The difference lies in the usage and the detailed structure. In the eager mechanism for materialized view update, a hypothetical relation represents the net changes made by a single transaction to a base relation and can be dropped after the transaction is committed and the materialized view is updated. In the continual query refresh method, a differential relation actually maintains changes made by several transactions to a base relation. Data in the differential relation can only be dropped when their timestamps are "older" than the timestamp of the latest execution of *every* relevant continual query.

We would like to mention that the relational model is not essential to our approach, but it simplifies the representation of database changes, allows use of the relational algebra, and avoids the need to explain the semantics of a particular object model.

[1] `Outerjoin` keeps all tuples in both the left and right relations when no matching tuples are found, padding them with `null` values as needed [28].

5.2.1 Differential Relations

We introduce the concept of differential relation, a relation that can represent changes to another relation, *and* design a set of basic operations to facilitate the manipulation of such relations. The goal for defining differential relations, instead of using hypothetical relations described in [5, 14], is to provide a unified treatment of changes, not separate treatments of insertions, deletions, and modifications resulting in several algorithms for generating and combining individual results.

Let $\mathcal{R}$ denote a relation scheme described by a set of attributes $A_1, A_2, \ldots, A_n$. Let R denote a relation instance of $\mathcal{R}$ consisting of a collection of tuples whose values are taken from the domains of the set of attributes of $\mathcal{R}$. $t.A_i$ $(1 \leq i \leq n)$ denotes the value of attribute A_i for tuple t. Each tuple has an attribute, denoted as *tid*, which provides a unique immutable identifier. When a tuple is deleted and later re-appended to R, it will have a new *tid* assigned. The unique tuple identifier *tid* makes it easier to connect tuples that hold values of the same object before and after changes. In fact, the primary key can be used as the unique identifier for each tuple of R.

Definition 1 (Differential relation)
Let $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ be a relation scheme. For each relation R of scheme $\mathcal{R}$, we define a differential relation, denoted by ΔR, to represent changes.

$$\Delta R = (tid^{<old>}, A_1^{<old>}; \ldots, A_n^{<old>}, tid^{<new>}, A_1^{<new>}, \ldots, A_n^{<new>}, \texttt{timestamp}),$$

where

- $A_i^{<old>}$ refers to old attribute values and
- $A_i^{<new>}$ $(1 \leq i \leq n)$ refers to new attribute values.
- For any tuple $t \in \Delta R$,
  - $tid^{<old>}$ and $tid^{<new>}$ cannot both be null.
  - If $tid^{<old>}$ is null, so are all $t.A_i^{<old>}$ $(1 \leq i \leq n)$.
  - Similarly, if $tid^{<new>}$ is null, so are all $t.A_i^{<new>}$ $(1 \leq i \leq n)$.
  - If both $tid^{<new>}$ and $tid^{<old>}$ are not null, then $tid^{<new>} = tid^{<old>}$.
- For each tuple in ΔR, the system assigns a `timestamp` at its creation time as its identifier.

Differential relations can represent tuples (or objects) where only the tid field is nonnull. No tid can appear in multiple rows. For addition, the attributes $A_i^{<old>}$ $(1 \leq i \leq n)$ are null. For deletion, the attributes $A_i^{<new>}$ are null. For modification, attributes $A_i^{<old>}$ hold old values and attributes $A_i^{<new>}$ hold newly modified values. The `timestamp` field is set to the current time (from a system clock, or any other monotonically increasing source of timestamps) whenever a tuple is appended to ΔR.

Example 4 Consider the relation **Stocks** with attributes such as name and price per 100 units.

Stocks:

tid	Name	Price
120992	DEC	150
092394	OLI	145
032090	ODI	120
041977	USL	100
⋮	⋮	⋮

Assume that the transaction **T** updates the **Stocks** relation by insertion, deletion and modification:

```
Transaction T {
    Insert (101088, MAC, 117);
    Modify (120992, DEC, 150) = (120992, DEC, 149);
    Delete (092394);
}
```

The following differential relation **ΔStocks** captures the changes that the transaction **T** made to **Stocks**:

ΔStocks:

$tid^{<old>}$	$Name^{<old>}$	$Price^{<old>}$	$tid^{<new>}$	$Name^{<new>}$	$Price^{<new>}$	timestamp
-	-	-	101088	MAC	117	10
120992	DEC	150	120992	DEC	149	25
092394	OLI	145	-	-	-	50

In the rest of this section, for presentation clarity the field **timestamp** is omitted when no confusion is incurred. We sometimes simply use "delta relations" to refer to differential relations.

5.2.2 Basic Operations

We first define renaming functions that add or remove the superscripts *old* and *new* from attribute names in a relation scheme $\mathcal{R}$.

Definition 2 Let $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ be a relation scheme and R be a relation of scheme $\mathcal{R}$. Let the relation S be of relation scheme $\mathcal{S} = (tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>})$. The differential relation schema of $\mathcal{R}$, is denoted by $\Delta\mathcal{R}$.

1. **old**$(\mathcal{R}) = \{tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>}\}$.
2. **new**$(\mathcal{R}) = \{tid^{<new>}, A_1^{<new>}, \ldots, A_n^{<new>}\}$.
3. **detach**$[S;$ **old**$(\mathcal{S})]$ returns the same relation with each attribute name detached from the superscript <old>.
4. **attach**$[R;$ **old**$(\mathcal{R})]$ returns the same relation with each attribute name attached by the superscript <old>.

The second group of operators is used to project the *old* or *new* values of a differential relation, and to compute the updated relation, say R', by combining the relation R with its differential relation ΔR.

Definition 3 (High-level projection operators)

Let R be a relation of $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ and ΔR be a relation of the differential relation schema $\Delta\mathcal{R}$. We define the operators `deletions`, `insertions` and `combine` as follows:

1. $\texttt{deleteLog}(\Delta R) = \Pi(\Delta R, \texttt{old}(\mathcal{R}))$.
2. $\texttt{insertLog}(\Delta R) = \Pi(\Delta R, \texttt{new}(\mathcal{R}))$.
3. $\texttt{deletions}(\Delta R) = \texttt{detach}[\texttt{deleteLog}(\Delta R); \texttt{old}(\mathcal{R})]$.
4. $\texttt{insertions}(\Delta R) = \texttt{detach}[\texttt{insertLog}(\Delta R); \texttt{new}(\mathcal{R})]$.
5. $\texttt{combine}(R, \Delta R) = (R - \texttt{deletions}(\Delta R)) \bigcup \texttt{insertions}(\Delta R)$.

These high-level projection operators are derivations of the basic operations for differential relations. We define them here mainly for presentation convenience, because they are used frequently in our differential re-evaluation strategies for continual query processing.

Example 5 Consider the relation `Stocks` and the differential relation ΔStocks in Example 4.

deleteLog(ΔStocks)

tid^{<old>}	Name^{<old>}	Price^{<old>}
120992	DEC	150
092394	OLI	145

insertions(ΔStocks)

tid	Name	Price
101088	MAC	117
120992	DEC	149

Stocks' = combine(Stocks, ΔStocks)

tid	Name	Price
120992	DEC	149
032090	ODI	120
041977	USL	100
101088	MAC	117
⋮	⋮	⋮

5.3 Differential Evaluation of Continual Queries

In this section we present a differential re-evaluation algorithm (DRA) for processing a continual query efficiently. In contrast to a complete re-evaluation, differential re-evaluation means that after the initial execution of a CQ, the re-evaluation of each subsequent execution of this CQ will be

performed by using the differential form of the query. The DRA is invoked by the CQ manager based on the epsilon specification associated with the given CQ. We assume that the information available when the DRA is invoked includes: (i)the CQ definition; (ii)the contents of each base relation after the last execution of the CQ; (iii)the differential relations for each of those operand relations that have been changed since the last execution of the CQ; (iv)the timestamp of the last execution of the CQ; (v)the complete set of the result of the CQ produced by the last execution. Note that the CQ manager will use (iv) to append the proper timestamp predicate(s) into the differential form of the CQ, which limits the search space to only those tuples that were written to the differential relations by the updates occurred after the last execution of this CQ (recall Example 6 for an illustration). We also formally study the correctness of the DRA with respect to the complete re-evaluation solution.

In what follows, we first formally define an operator that computes the differential relations for the data derived by arbitrary query expressions in Section 5.3.1. In Section 5.3.2 we introduce the differential forms for the three basic relational algebraic operations: **Select**, **Project**, and **Join**. We prove that instantiation of **Propagate**(Q; PL) for relational select, project, and join are functionally equivalent to their differential forms: **DiffSelect**, **DiffProj** and **DiffJoin**. The differential re-evaluation algorithm is described in Section 5.3.3.

5.3.1 Computing the Differential Results for Continual Queries

In this section we introduce a high level operator, called **Propagate**, to describe how the result relation of a continual query changes when at least one of its operand relations changes. This operator computes the difference between two consecutive executions of a CQ by complete re-evaluation of the query for each execution. It can be considered as an instantiation of *complete re-evaluation* solution. The main purpose of introducing the operator **Propagate** is to formally prove that our differential re-evaluation algorithm to continual queries is *functionally equivalent* to the "recompute the query from scratch" solution, and, in many situations is more efficient.

Before giving the definition of the operator **propagate**, we define an operator that compute the difference of two relations of the same scheme.

Definition 4 (The operator **Diff**)
Let R_1 and R_2 be two relations of the same scheme $\mathcal{R}$. The operator $\mathtt{Diff}(R_1, R_2)$ is defined as the $\mathtt{Outerjoin}_{\mathtt{tid}}$ of tuples in $R_1 - R_2$ and $R_2 - R_1$ under the join condition "$\mathtt{tid}^{<old>} = \mathtt{tid}^{<new>}$".

$$\mathtt{Diff}(R_1, R_2) = \mathtt{Outerjoin}_{\mathtt{tid}}(\mathtt{attach}[(R_1 - R_2);\mathtt{old}(\mathcal{R})], \mathtt{attach}[(R_2 - R_1);\mathtt{new}(\mathcal{R})]).$$

It returns a differential relation ΔR that describes their differences. Null values are used to pad tuples that appear in only $R_1 - R_2$ or $R_2 - R_1$. The relation scheme of ΔR is $\mathtt{old}(\mathcal{R}) \cup \mathtt{new}(\mathcal{R})$.

Definition 5 (The **Propagate** operator)
Let R_i be a relation of scheme $\mathcal{R}_i$ $(1 \leq i \leq n)$, ΔR_i be a differential relation of R_i, and R_i' denote

combine$(R_i, \Delta R_i)$. Let $Q(R_1, \ldots, R_n)$ denote an arbitrary continual query and SL = $\{[R_i, \Delta R_i] \mid (1 \leq i \leq k, k \leq n)\}$ denote the differential substitution list. The **Propagate** operator is defined as follows:

$$\textbf{Propagate}(Q(R_1, \ldots, R_n); \text{SL}) = \textbf{Diff}(Q(R_1, \ldots, R_n), Q(R'_1, \ldots, R'_n)) \qquad (n \geq 1).$$

It returns the differential relation of query Q, denoted by ΔR_Q, with the scheme $\mathcal{R}_Q$.

We may also denote **Propagate**$(Q(R_1, \ldots, R_n); \text{SL})$ by $\Delta Q(R_1, \ldots, R_n)$.

Example 6 Consider the query Q: $\sigma_{\texttt{price}>120}(\texttt{Stocks})$ as a continual query. Let E_i be the ith execution of Q at time t_i and $E_i(Q, t_i)$ denote the result of the ith execution of Q ($i = 0, \ldots, \infty$). Now assume that the base relation **Stocks** is changed, after the last execution E_i and before the current execution E_{i+1}, by the update transaction **T** given in Example 4. Let Q(**Stocks**) = $E_i(Q, t_i)$ denote the result of the ith execution of Q over **Stocks**. Let **Stocks'** denote the base relation after updates to **Stocks** by transaction **T**, and Q(**Stocks'**) = $E_{i+1}(Q, t_{i+1})$ denote the result of the current execution of Q, over the relation **Stocks'**.

Based on Definition 5, to express how the result $E_i(Q, t_i)$ may change after the updates by the transaction **T**, we may simply compute **Propagate**(Q(**Stocks**);[**Stocks**,Δ**Stocks**]), the difference between the result relation before the updates: Q(**Stocks**) and the result relation after the update: Q(**Stocks'**).

Q(**Stocks**) = $\sigma_{\texttt{price}>120}$(**Stocks**) = {(120992,DEC,150), (092394,OLI,145)}.

Q(**Stocks'**) = $\sigma_{\texttt{price}>120}$(**Stocks'**) = {(120992,DEC,149)}.

Q(**Stocks**)−Q(**Stocks'**) = {(120992,DEC,150), (092394,OLI,145)}.

Q(**Stocks'**)−Q(**Stocks**) = {(120992,DEC,149)}.

The differential result ΔQ(**Stocks**) below represents the net effect made by all the updates occurred between the the last execution E_i and the current execution E_{i+1}.

ΔQ(**Stocks**) = **Diff**(Q(**Stocks**),Q(**Stocks'**))

tid$^{<old>}$	Name$^{<old>}$	Price$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$	Price$^{<new>}$
120992	DEC	150	120992	DEC	149
092394	OLI	145	-	-	-

In this example, the differential result of the query $\sigma_{\texttt{price}>120}$(**Stocks**), since the last execution of Q, is computed by directly evaluating the function: **Propagate**($\sigma_{\texttt{price}>120}$(**Stocks**);[**Stocks**,Δ**Stocks**]). Assume the previous execution result $E_i(Q, t_i)$ is saved. The evaluation of **Propagate** directly from its definition requires first a scan of the modified relation **Stocks'** in order to compute Q(**Stocks'**). Such recomputing from scratch is often wasteful, and in many cases unacceptable.

Observe that when (i)the size of the source relation **Stocks** is large, (ii)the selectivity factor of the query Q over **Stocks** is not high, and (iii)the number of tuples written by the updates, occurred between the two consecutive executions E_i and E_{i+1}, is relatively small, it would be more efficient to compute the differential result of the query Q before and after the updates

by evaluating the query over the differential relation **ΔStocks** instead. It is because computing **Propagate**($\sigma_{\text{price}>120}$(**Stocks**);[**Stocks**,**ΔStocks**]) is functionally equivalent to the evaluation of the differential query: σ_F(**ΔStocks**), where F denotes the predicate **price**$^{<old>}$ $> 120 \wedge$ **price**$^{<new>} > 120 \wedge$ **timestamp**$> t_i$[1]. This is true even when the user needs the complete composite set of the query result, because computing the union of Q(**Stocks**) and **insertions**(σ_F(**ΔStocks**)) is cheaper than recomputing the expression $\sigma_{\text{price}>120}$(**Stocks'**) from scratch.

Furthermore, using a differential evaluation approach, we can show those tuples that were removed between the two consecutive executions of Q, simply by computing **deletions**(σ_F(**ΔStocks**)). In general, for any continual query Q over the relation R, let $E_i(Q,t_i)$ be the last execution of Q at time t_i. A complete set of the result of current execution of Q can be obtained by computing the expression:

$$(E_i(Q,t_i) - \sigma_{timestamp>t_i}(\textbf{deletions}(\Delta R))) \cup \sigma_{timestamp>t_i}(\textbf{insertions}(\Delta R)).$$

The expression $\sigma_{timestamp>t_i}$(**insertions**(ΔR) returns all the records that have been appended to R since the last execution of Q; whereas the expression $\sigma_{timestamp>t_i}$(**deletions**(ΔR) returns all the records removed from R since the last execution of Q.

5.3.2 Optimization based on Differential Operators

For many forms of queries, **Propagate**(Q; SL) can be computed more efficiently than by directly evaluating the definition of the **Propagate** operator. These computations use the differential relations heavily and sometimes exclusively, instead of computing on the base relations, which tend to be much larger.

In this section we define the differential forms for the three basic relational algebraic operations: **Select**, **Project**, and **Join**. We prove that instantiation of **Propagate**(Q; SL) for relational select, project, and join are functionally equivalent to their differential forms: **DiffSelect**, **DiffProj** and **DiffJoin**. The predicates contained in these operators can be atomic or composite by logical "**AND**" and logical "**OR**". The attributes involved in the predicates or the projection list are single valued attributes. Aggregate functions such as **SUM**, **COUNT**, **MAX**, **MIN** are allowed in this framework with some additional consideration. For example, when the query expression contains the aggregate function **MAX**(A), we need to compare the A field of each $< new >$ tuple in the differential relation with the value of **MAX(A)**, and to reset **MAX(A)** if necessary. Similar treatment applies to **SUM**, **COUNT**, **MIN**. However when the query contains the aggregate function **AVG**, the recomputation of the query is needed.

5.3.2.1 The Differential Select Operator

Definition 6 (Differential select)
Let F denote a predicate defined on the relation R of scheme $\mathcal{R}$, and $\sigma[R;F]$ denote the associated

[1] The condition **timestamp**$> t_i$ is appended by the CQ manager for incorporating correct amount of updates and limiting the query search space. Section 5.5.3 provides some further discussion.

relational selection operator. Let $F^{<old>}$ and $F^{<new>}$ denote the same predicate with all attribute names superscripted accordingly. We define the operator **DiffSelect** as follows:

$$\texttt{DiffSelect}[\Delta R;F] = \texttt{Outerjoin}_{tid}(\sigma[\texttt{deleteLog}(\Delta R);F^{<old>}], \sigma[\texttt{insertLog}(\Delta R);F^{<new>}]).$$

Proposition 1 $\texttt{Propagate}(\sigma[R;F];[R,\Delta R]) \equiv \texttt{DiffSelect}[\Delta R;F]$.

The formal proof of this proposition is omitted here. Readers may refer to [24] for further detail. This proposition shows that Definition 6 gives an implementation of **DiffSelect** which does not reference the base relation, and which can be implemented by one pass over ΔR, determining for each tuple of ΔR whether it induces an insertion, deletion, or modification to $\sigma[R;F]$.

When $|R| \geq |\Delta R|$, it is cheaper to re-evaluate the query expression $\sigma_P(R)$ by using the differential select operator $\texttt{DiffSelect}(\Delta R; P)$ rather than recomputing the expression $\sigma_P(R)$ from scratch.

5.3.2.2 The Differential Project Operator

Definition 7 (Differential project)
Let R be a relation (base or derived) of scheme $\mathcal{R} = (A_1, \ldots, A_n)$. Let $\mathbf{X} = \{A_{i_1}, \ldots, A_{i_k}\}$ denotes a subset of the attributes in $\mathcal{R}$, $\texttt{tid} \in \mathbf{X}$, and $i_j \in \{1, \ldots, n\}$ $(1 \leq j \leq k)$. Let P denote the predicate $\bigvee_{j=1}^{k}(A_{i_j}^{<old>} \neq A_{i_j}^{<new>})$ if R is a (persistent) base relation; otherwise P denote the truth value **true**. The differential project operator **DiffProj** is defined as follows:

$$\texttt{DiffProj}[\Delta R;\mathbf{X}] = \sigma_P(\Pi[\Delta R;\ \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]).$$

The following proposition shows that **DiffProj** is an efficient differential form that can be used to implement $\texttt{Propagate}(\Pi[R;\mathbf{X}];(R,\Delta R))$, because they are functionally equivalent, **DiffProj** runs one pass over ΔR, and the **Propagate** runs over R' which tends to be much larger than ΔR.

Proposition 2 $\texttt{Propagate}(\Pi[R;\mathbf{X}];(R,\Delta R)) \equiv \texttt{DiffProj}[\Delta R;\ \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]$.

Similarly, the formal proof of this proposition is omitted here. Readers may refer to [24] for further detail.

Example 7 Consider the following two queries over the relation **Stocks** in Example 4:

$$Q_1 = \Pi[\texttt{Stocks};\ \texttt{Name}].$$
$$Q_2 = \Pi[\sigma_{\texttt{price}>120}(\texttt{Stocks});\ \texttt{Name}].$$

Let $\texttt{HighStocks} = \sigma(\texttt{Stocks};\ \texttt{Price>120})$. By Definition 5, the evaluation of $\texttt{Propagate}(\sigma_{\texttt{price}>120}(\texttt{Stocks});\ [\texttt{Stocks},\Delta\texttt{Stocks}])$ requires as the inputs both the base relation **Stocks** before the updates and the base relation after the updates, which is **Stocks'**. According to

Proposition 2 and Definition 7, it would be more efficient to process both queries (Q_1 and Q_2) using the differential operator **DiffProj** than directly using the definition of **Propagate**.

- **Propagate**(Π[**Stocks**; **Name**]; [**Stocks**,Δ**Stocks**])
  = **DiffProj**[Δ**Stocks**; **Name**].

tid$^{<old>}$	Name$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$
-	-	101088	MAC
092394	OLI	-	-

- **Propagate**(Π[**HighStocks**; **old**(**Name**)∪**new**(**Name**)]; [**HighStocks**,Δ**HighStocks**])
  = **DiffProj**[Δ**HighStocks**; **old**(**Name**)∪**new**(**Name**)]
  = **DiffProj**[**Propagate**(**HighStocks**;[**Stocks**,Δ**Stocks**]); **old**(**Name**)∪**new**(**Name**)]
  = **DiffProj**[**DiffSelect**(Δ**Stocks**); **old**(**Name**)∪**new**(**Name**)].

tid$^{<old>}$	Name$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$
120992	DEC	120992	DEC
092394	OLI	-	-

5.3.2.3 The Differential Join Operator

We first consider the join over two relations R_1 and R_2 (i.e., $n = 2$). In this case changes to the resulting relation can be induced by changes to either input operand or both of them. Therefore, the differential join operator **DiffJoin** should consider the following three cases when computing the total changes to the result relation of $R_1 \bowtie R_2$: (i)only R_1 changes; (ii)only R_2 changes; and (iii)both R_1 and R_2 change.

Definition 8 (Differential join)
Let P_{join} be a predicate on $\mathcal{R}_1$ and $\mathcal{R}_2$, and let $\mathbf{tid}_1 \in \mathcal{R}_1$ and $\mathbf{tid}_2 \in \mathcal{R}_2$ respectively. Let $P_{join}^{<old>}$ and $P_{join}^{<new>}$ denote the predicates obtained from P_{join} by attaching each attribute name with superscript $< old >$ and $< new >$ respectively. Let $P(tid_1, tid_2)$ denote the predicate "$(tid_1^{<old>} = tid_1^{<new>})$ **AND** $(tid_2^{<old>} = tid_2^{<new>})$". Let $\mathcal{R}$ denote $\mathcal{R}_1 \cup \mathcal{R}_2$. We define the operator **DiffJoin** as follows:

$$\mathbf{DiffJoin}_{P(tid_1,tid_2)}(R_1, R_2)$$
$$= \mathbf{DJoin}_{P_{join}}(\Delta R_1, R_2) \bigcup \mathbf{DJoin}_{P_{join}}(R_1, \Delta R_2) \bigcup \mathbf{DJoin}_{P_{join}}(\Delta R_1, \Delta R_2).$$

where

- $\mathbf{DJoin}_{P_{join}}(\Delta R_1, R_2)$
  $= \mathbf{Outerjoin}_{P(tid_1,tid_2)}(\mathbf{attach}[\bowtie_{P_{join}^{<old>}}(\mathbf{deletions}(\Delta R_1),\ R_2);\ \mathbf{old}(\mathcal{R})],$
  $\mathbf{attach}[\bowtie_{P_{join}^{<new>}}(\mathbf{insertions}(\Delta R_1),\ R_2);\ \mathbf{new}(\mathcal{R})]).$
- $\mathbf{DJoin}_{P_{join}}(R_1, \Delta R_2)$
  $= \mathbf{Outerjoin}_{P(tid_1,tid_2)}(\mathbf{attach}[\bowtie_{P_{join}^{<old>}}(R_1,\ \mathbf{deletions}(\Delta R_2);\ \mathbf{old}(\mathcal{R})],$
  $\mathbf{attach}[\bowtie_{P_{join}^{<new>}}(R_1, \mathbf{insertions}(\Delta R_2);\ \mathbf{new}(\mathcal{R})]).$

- $\texttt{DJoin}_{P_{join}}(\Delta R_1, \Delta R_2)$

  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\bowtie_{P_{join}^{<old>}}(\texttt{deleteLog}(\Delta R_1), \texttt{deleteLog}(\Delta R_2)),$

  $\bowtie_{P_{join}^{<new>}}(\texttt{insertLog}(\Delta R_1), \texttt{insertLog}(\Delta R_2)).$

Proposition 3 Let the differential substitution list SL be $\{[R_1, \Delta R_1],[R_2, \Delta R_2]\}$.

$\texttt{Propagate}(\bowtie_{P_{join}}(R_1, R_2); \{[R_1, \Delta R_1],[R_2, \Delta R_2]\}) \equiv \texttt{DiffJoin}_{P(tid_1,tid_2)}(R_1, R_2)$.

This development can be generalized to the join of an arbitrary number of base relations. Let $Q = R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n$ $(n \geq 2)$ denote an arbitrary join expression. The equation $\texttt{Propagate}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n; SL) \equiv \texttt{DiffJoin}_{P_{tid_1,\ldots,tid_n}}(R_1, R_2, \ldots, R_n)$ holds in general. Also, when the number k $(k \leq n)$ relations have been changed since the last execution of Q, to evaluate $\texttt{DiffJoin}_{P_{tid_1,\ldots,tid_n}}(R_1, R_2, \ldots, R_n)$, we need to consider only $2^k - 1$ cases, each representing one type of change effects. The total changes to the result of Q will be the union of these cases. According to the associative and symmetric property of relational join, we may assume that the first k relations (i.e., $R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_k$, $k \leq n$) are those that have been changed without loss of generality.

By associating a truth table T, with k columns and $p = 2^k - 1$ rows, to the query Q, it is easy to compute each possible combination of joins, which needs to be considered when computing the changes to the result of Q after k operand relations have been changed. Each column of the T table corresponds to an updated relation in Q since the last execution of Q. Each row represents one possible case that the computation of $\texttt{DiffJoin}(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n)$ considers. The formal proof of this proposition is also omitted here. Readers may refer to [24] for detail.

Example 8 For a query $Q = R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n$ $(n \geq 2)$, consider the case when $k = 3$ and $k < n$. Based on the T table associated with Q below, we need to consider only seven cases: (1)R_1, R_2, R_3 all change; (2)only R_1, R_2 change; (3)only R_1, R_3 change; (4)only R_2, R_3 change; (5)only R_1 changes; (6)only R_2 changes; (7)only R_3 changes. Each row in the T table corresponds to one possible combination of join required to compute the changes to the result of Q.

T1	T2	T3	$\texttt{DiffJoin}(R_1, R_2, R_3, \ldots, R_n) =$
1	0	0	$\texttt{DJoin}(\Delta R_1, R_2, R_3, \ldots, R_n)$
0	1	0	$\cup\, \texttt{DJoin}(R_1, \Delta R_2, R_3, \ldots, R_n)$
0	0	1	$\cup\, \texttt{DJoin}(R_1, R_2, \Delta R_3, \ldots, R_n)$
1	1	0	$\cup\, \texttt{DJoin}(\Delta R_1, \Delta R_2, R_3, \ldots, R_n)$
1	0	1	$\cup\, \texttt{DJoin}(\Delta R_1, R_2, \Delta R_3, \ldots, R_n)$
0	1	1	$\cup\, \texttt{DJoin}(R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$
1	1	1	$\cup\, \texttt{DJoin}(\Delta R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$

Let $\mathcal{R}_i$ be the scheme of relation R_i $(1 \leq i \leq 3)$ and $\mathcal{R}$ denote $\cup_{i=1}^{n} \mathcal{R}_i$. The definitions for the above DJoins are similar to those in Definition 8. For instance,

$\texttt{DJoin}_{P_{join}}(\Delta R_1, R_2, R_3, \ldots, R_n)$

$= \texttt{Outerjoin}_{tid}(\texttt{attach}[\texttt{deletions}(\Delta R_1) \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n); \texttt{old}(\mathcal{R})],$

$\texttt{attach}[\texttt{insertions}(\Delta R_1) \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n); \texttt{new}(\mathcal{R})]).$

$$\texttt{DJoin}_{P_{join}}(\Delta R_1, R_2, \Delta R_3, \ldots, R_n)$$
$$= \texttt{Outerjoin}_{tid}(\texttt{attach}[\texttt{deletions}(\Delta R_1) \bowtie R_2 \bowtie \texttt{deletions}(\Delta R_3) \bowtie \ldots \bowtie R_n); \texttt{old}(\mathcal{R})],$$
$$\texttt{attach}[\texttt{insertions}(\Delta R_1) \bowtie R_2 \bowtie \texttt{insertions}(\Delta R_3) \bowtie \ldots \bowtie R_n); \texttt{new}(\mathcal{R})]).$$

Observe that the above example exhibits an interesting optimization problem, namely, the efficient execution of a set of n-ary join expressions in which intermediate results can be reused among several *SPJ* expressions. For instance, when $n > 4$ in the above query Q, let $W_1 = R_4 \bowtie \ldots \bowtie R_n$ and $W_2 = R_2 \bowtie R_3 \bowtie W_1$. Saving W_1 and W_2 as intermediate results, and then re-using them in the evaluation of each of the seven **DJoin** expressions above, we may easily speed up the processing of **DiffJoin**. This mechanism works effectively when n is larger than k.

The idea of using the truth table to facilitate the combination of possible joins was borrowed from the research in updating materialized view [5, 14]. We minimize the cost of constructing such a table by using as the columns of the table only the number of changed relations, instead of the n operand relations in the query expression $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n))$.

Moreover, in the DIOM system, we apply several conventional query optimization techniques used in both centralized and distributed environment to further reduce the cost of **DiffJoin** operation. For example, given a continual query Q, denoted by $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_{50}))$, which request access to 50 classes/relations from 20 different information sources. Assume only two relations (say R_1 and R_2) have been updated since the last execution of the query Q. By using the commutativity of selection and projection over joins and associativity of joins, this query Q will first be decomposed into the following two subqueries: $SubQ_1 = \Pi_{X_1}(\sigma_{F_1}(R_3 \bowtie R_4 \bowtie, \ldots, \bowtie R_{50}))$ and $SubQ_2 = \Pi_{X_2}(\sigma_{F_2}(R_1 \bowtie R_2 \bowtie Result(SubQ_1)))$, where $X = X_1 \cup X_2$ and F_1 is a selection condition over $R_3, R_4, \ldots, R_{50}$ and F_2 is a selection condition over R_1, R_2, and the result of $SubQ_1$. Now the evaluation of the **DiffJoin** operator over Q is reduced to **DiffJoin** over the subquery $SubQ_2$. The system performance for processing this CQ will be greatly improved, because (i)the evaluation of $SubQ_1$ can be done directly against the previous execution result of Q cached at the client side, and (ii)comparing with the original query Q, the size of $R_1 \bowtie R_2 \bowtie Result(SubQ_1)$ is much smaller both in cardinality and in degree than $(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_{50})$. This approach is particularly benefical when the selectivity of F_1 is high and the project list X_1 is small.

5.3.3 The Differential Re-evaluation Algorithm

We now outline an algorithm for re-evaluating continual queries (limited to SPJ expressions) differentially.

Algorithm 1 (The DRA algorithm)

Input:

- the SPJ definition of the continual query Q, i.e., $Q = \Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n))$, where X denotes the projection list and F denotes the selection predicate over $R_1, \ldots, R_n$;

- the contents of the base relations R_i $(1 \leq i \leq n)$ after the last execution of the CQ;
- the differential relations ΔR_i $(1 \leq i \leq n)$;
- the timestamp of the last execution of this CQ, say E_i;
- the complete set of the result produced by the last execution of the CQ.

Output: the result of the current execution of the query Q.

Procedural Steps:

1. Build the truth table T with k columns ($k \leq n$) and p rows, $p = 2^n$. Each column is corresponding to a relation in the SPJ expression, which has been changed since the last execution of Q.
2. For each row i $(1 \leq i \leq p)$ of the table T, construct the associated SPJ expression, by substituting R_i in Q with ΔR_i when the binary variable $T_{ij} = 1$. For each of these SPJ expression, denoted by $G = S_1 \bowtie S_2 \bowtie, \ldots, \bowtie S_n$, evaluate G by its differential form $\texttt{DJoin}(S_1 \bowtie S_2 \bowtie, \ldots, \bowtie S_n)$.
3. Perform the union of the results obtained from each computation in Step 2.
4. Based on the epsilon specification of the CQ, assemble the final set of the result to be returned to the users.

For example, let ΔR_Q denote the result generated by Step 3.

- If the user wishes to see only the differential result since the last execution of the Q, say $E_i(Q, t_i)$, without deletion notification, the result to be returned can be computed by $\sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R_Q))$.
- If the user needs to see the complete set of the result matching the query, we return $E_i(Q,t_i) \bigcup \sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R_Q))$.
- If the user wants to be notified all the deleted tuples since the last execution of the CQ, we simply compute the $\sigma_{timestamp>t_i}(\texttt{deletions}(\Delta R_Q))$.

We use the relational model to describe the DRA algorithm. This is a design choice. In principle, CQs could be written in query languages that assume other data models. In this chapter, we do not address the issue of query language translation, which by itself is a significant research topic. The extension of the DRA algorithm to object-oriented models will become important when more data become available on object-oriented databases, and more queries are written in object-oriented query languages. At present, most of organized data are stored in relational database management systems and queries written in SQL. Similarly, most of unorganized data (such as WWW pages) are stored in files and queries over these data are submitted through simple GUIs and can be translated into SQL.

5.4 Processing Continual Queries: Simple Examples

In this section, we use examples to illustrate the differential evaluation based on differential relations (log data) for processing continual queries. We also compare our approach with the timestamp-based transformation strategy [37], and demonstrate the benefits of using differential evaluation strategy in a database environment WHERE data items can be appended, removed, or modified dynamically.

Example 9 Suppose the user wants to install the following two queries as continual queries:

```
Q1:  SELECT * FROM Stocks
     WHERE price < 120

Q2:  SELECT * FROM Stocks
     WHERE name = "DEC" OR name = "MAC"
```

(1) Using the differential evaluation based on differential relations, we may easily transform the above queries into the queries over the differential relation ΔStocks as follows:

```
IQ1:  SELECT * FROM ΔStocks WHERE price < 120

IQ2:  SELECT * FROM ΔStocks WHERE name = "DEC" OR name = "MAC"
```

The initial execution of Q1 returns (101088, USL, 100). The initial execution of Q2 returns (120992, DEC, 150). After the initial execution of a continual query, say Q1 or Q2, the subsequent executions of the same query will be carried out by the differential query IQ1 or IQ2. Suppose now the database is changed by the transaction T described in Example 4. According to the Stocks' relation given in Example 5, two tuples (101088, USL, 100) and (101088, MAC, 117) are qualified for Q1. The first one remains the same and the second one is a new tuple inserted by T. The execution of the differential query IQ1 guarantees that only the newly added tuple (101088, MAC, 117) is returned. When the base relation is large in size, using the differential query evaluation based on differential relations will drastically reduce the computation cost of the continual queries. Similarly, executing Q2 as a continual query after transaction T is committed, the tuples (120992, DEC, 149) and (101088, MAC, 117) will be returned by IQ2.

(2) Using the timestamp based transformation approach proposed by [37], two preconditions must be satisfied:

- The database is restricted to append only in the sense that data items are appended to the database as they arrive and are never removed or modified [37].
- Each object instance (relation tuple) should have a timestamp associated with to represent its creation time.

The queries Q1 and Q2 will first be transformed to time-stamped queries by adding timestamp related conditions to the query expression, for example:

```
MQ1: SELECT * FROM Stocks p
     WHERE price < 120 AND p.st < CURRENT_TIMESTAMP

MQ2: SELECT * FROM Stocks p
     WHERE name = "DEC" OR name = "MAC"
     AND   p.st < CURRENT_TIMESTAMP
```

If the the system may provide the timestamp, say d, of the previous execution of a continual query, then the above time-stamped queries may be further rewritten to reduce the search space.

```
MQ1: SELECT * FROM Stocks p
     WHERE price < 120 AND p.st > d

MQ2: SELECT * FROM Stocks p
     WHERE name = "DEC" OR name = "MAC"
     AND   p.st > d
```

In our view, adding timestamp to every tuple of the existing legacy systems is far more expensive than maintaining differential relations, especially when the database is large in size. Also many real-world applications, for which continual queries would be a useful tool, allow data items to be appended, removed or modified at any time.

5.5 Discussion

In previous sections we have defined the differential forms for the three basic relational algebraic operations: **Select**, **Project**, and **Join**, and developed a differential re-evaluation algorithm (DRA). The key idea of the DRA method is to transform a continual query over the source data (base relations) into a differential query that runs over the corresponding delta relations. As formally proved in [24], for any SPJ expression, using differential re-evaluation method is functionally equivalent to the complete re-evaluation of the query.

In this section we discuss a number of issues related to performance optimization opportunities for continual query processing.

5.5.1 Strawman Performance Arguments

Although there is no space in this chapter for a detailed performance analysis, we argue informally that there are many important scenarios in which DRA wins over algorithms that operate on the base data instead of results.

First of all, we observe that the overwhelming majority of queries return a table of results that is much smaller than the base data. (Otherwise, the query would be considered not selective enough and the results not particularly useful.) Therefore, DRA processing of the next query execution on top of results will be much faster, reducing both I/O and CPU requirements and communication overhead. In general, caching the results on the client side makes the servers more scalable with respect to the number of clients.

Second, since results are combined from many sources into a local table, DRA processing of results will avoid both translation from the base data to an interoperable format. Moreover, if the volume of relevant updates is smaller than the results (which is the common case), then we are further reducing the network traffic.

Third, each server only generates delta relations when communicating with the clients. This is easier for interoperation than trying directly to integrate active databases and materialized views. To the best of our knowledge, there are no practical methods for combining them in a heterogeneous environment.

On the other hand, we note some limitations of the DRA algorithm. For example, when the results turn out to be large (poor selectivity of the query), then a lazy evaluation and transmission of results is necessary. Another important assumption of the DRA algorithm is the availability of delta relations from every information producer. This may not be trivial for legacy databases. But as we mentioned before, there is no easy way to integrate legacy active databases or materialized views, either.

5.5.2 Query Refinement

First, we should test the CQ condition based on the differential updates before every execution. If the updates occurred in between of the two consecutive executions have no impact on the previous query result set, we consider them as irrelevant updates to the continual query. Thus, no computation is performed for this CQ, because in this case, nothing needs to be returned if the user is only interested in the differential result. When the user asks for the complete answer, we simply return the result of previous execution.

In addition, for each SPJ expression in step 2 of the DRA algorithm, it is necessary to determine its execution strategy. One way to find a good execution strategy is simply to use the heuristics such as `Select` before `Join`, extracting common subexpressions, cheaper selection predicate before expensive ones). This approach might be most appropriate if one does not have access to an appropriate query optimizer. An alternative approach is to have a DBMS query optimizer generate the strategies. The differential form of a query can be regarded as a query to a database that consists of base relations and differential relations.

5.5.3 Garbage Collection of Differential Relations

As the source data changes, their differential relations grow accordingly. To keep the differential relations to a bounded size, we need to garbage collect the portions of the differential relations that are no longer useful. The technical details of the solution are beyond the scope of this paper. We outline the basic idea here. First let us consider the case of a single active CQ in the system. Each time a new query result $Q(S_i)$ is obtained, we can retire the differential relations referring to database states prior to t_i. This is intuitively easy to understand since only the data in the differential relations with the timestamps later than t_i will be needed for the processing of $Q(S_{i+1})$. Informally, we call these portions of the differential relations "active delta zone".

With multiple active CQs in the system, the garbage collection algorithm is an extension of the basic idea outlined above. For each CQ, we define its active delta zone. For the whole system, we define the system active delta zone as the union of the active delta zones of all CQs. Assuming that each CQ will make progress, its active delta zone will move forward in time. The system active delta zone will move forward as a consequence, with its boundary delimited by the "oldest" active delta zone. All the data in the differential relations that fall outside the system active delta zone can be garbage collected, since they will not be used by any active CQ.

5.5.4 Generation of Delta Relations

It should be pointed out that, although we use the relational model terminology and concepts in the design and description of the DRA for clarity and simplicity, the DRA itself takes as input the updates from different information sources. These updates are described as differential relations in this paper, as differential relations have a very simple and clear form and content for representing updates in terms of modifications, insertions, and deletions. For the relational information source providers, the generation of different relations is quite straightforward. For those information sources other than relational databases, simple translators (as part of the DIOM services [20]) will be used to catch the updates in the form of differential relations. For example, file system updates can be captured by either operating system or middleware and translated into a differential relation and fed into DRA. This is in contrast to the conceptual difficulties in the integration of active databases and view materialization, as well as the practical difficulties of implementing these powerful database techniques in non-database environments such as file systems.

5.6 Implementation Consideration for DRA

The DRA implementation is under development in the prototype CQ system. It has not reached the working stage yet. However, some tools and utilities have already been implemented for the DRA development. The description of the available tools and future implementation plan is listed below:

- Step 1:

  Automatically generate delta tables for objects of a particular data source, which capture the updates on the source objects .

  A wrapper function has been implemented in Perl to facilitate auto-generation of delta tables for target data objects.[1] The code for the delta table generation tool is shown in Figure 5-1.

  For example, if we want to create the delta table for the source object table "STOCK" at data source "FINANCIAL", given the structure of "STOCK" as follows:

SYMBOL	PRICE

  A delta relation with the name "DELTA_FINANCIAL_STOCK" will be created (*tid* is not necessary since it can be derived from the key attributes):

OLD_SYMBOL	OLD_PRICE	NEW_SYMBOL	NEW_PRICE	TIMESTAMP

  This delta table will record all the updates on the stock objects performed locally at the data source. The CQ server will make use of this delta table to execute differential queries on "STOCK" object.

- Step 2:

  Given a continual query $(Q, T_{cq}, \texttt{Stop})$, transform the query component Q into an algebraic query graph with selection, projection, join operators as the internal nodes and data source objects as the leaf nodes.

- Step 3:

  Design and implement delta query operator for each primitive algebraic operations such as $DiffSelect$, $DiffProj$, and $DiffJoin$.

- Step 4:

  Design and implement a generic delta query transformation module which maps an arbitrary query in algebraic expression into a delta query expression.

- Step 5:

  Design and implement the refresh strategies to keep the delta relations up to date with the state of data sources. There will be a system component (we call it *Refresher*) running continually in background, which is independent of the continual query component. Basically, the *Refresher* will insert into the delta relations the new updates at the data source.

[1] The data objects from the target data source are first mapped to the continual query system object model, which is a relational model as in RDBMSs.

- Step 6:

  Design and implement a garbage collection module for delta relations. Since the delta relations grow whenever the *Refresher* updates them. Consequently, delta relations may become larger than the base tables. As a matter of fact, the objects in the delta table may not be useful any more when there is no continual query in the system that is using them(recall Section 5.5.3). Thus we have implemented a *Garbage Collector* which removes those delta objects that are out of date. More concretely, when the timestamp T_1 of a record r_1 in the delta relation DELTA_SRC_OBJ is older than the last execution timestamp of a particular *continual query*, it means that the corresponding update of the source object must have been accounted for during the last execution of the *continual query*. Therefore, record r_1 is considered "garbage" for the particular *continual query*. When T_1 is older than all the timestamps of the registered continual queries related with the object SRC.OBJ, record r_1 can be deleted from the delta relation. The *Garbage Collector* is also scheduled to be running periodically.

```
#!/usr/local/bin/perl
################################################################################
#Delta tables auto-generation program
#generate delta tables in CQ meta database
#call convention:
#createDelta.pl source_name source_loginname {src_tbname[,src_tbname]}
################################################################################
use Oraperl;
require "dbenv.pl";

$PREOLD = "OLD";          #prefix for old columns in delta tables
$PRENEW = "NEW";          #prefix for new columns in delta tables
$CREATEFILE = "createDelta.sql";#output SQL file for delta table creation
$DROPFILE = "dropDelta.sql";    #output SQL file for delta table deletion
main:
{
  $source = @ARGV[0];
  $src_login = @ARGV[1];
  shift @ARGV;
  shift @ARGV;
  $passwd = $PASSTAB{$srclogin};
  $metadb_login = "CQ";
  $metadb_passwd = $PASSTAB{$metadb_login};

  open(CREATE,"> $CREATEFILE");
  open(DROP,"> $DROPFILE");

  #get the names of all the source tables
  #$createStr stores the SQL statement to create the delta table
  foreach $tbname (@ARGV) {
    $createStr = "CREATE TABLE DELTA_$source_$tbname(
                  timestamp DATE \n";

    $query = "select * from $tbname";
    $csr = &runQuery($source, $src_login, $src_passwd, $query) || &dberror;
    #get the names of the source table columns
    @colsTitle = &ora_titles($csr, 0);
    #get the data types of the source table columns
    @types = &ora_types($csr);

    $numOfCols = $#colsTitle + 1;
    for ($i = 0; $i < $numOfCols; $i++) {
      $createStr .= "${PREOLD}_$colsTitle[$i] ";
      $tmp = "$oratypes{$types[$i]},";
      $createStr .= $tmp."\n";
      $createStr .= "${PRENEW}_@colsTitle[$i] ".$tmp."\n";
    }
    chop($createStr); #erase the last "\n"
    chop($createStr); #erase the last ","
    #add the finish parenthesis for the CREATE TABLE SQL statement
    $createStr .= ")";

    &runQuery($source, $src_login, $src_passwd, $createStr) || &dberror;
    print CREATE "${createStr}; \n\n";
    print DROP "drop table ${delta_dbname}.DELTA_$tbname; \n";
  }
  close(CREATEFILE);
  close(DROPFILE);
}
```

Figure 5-1: Delta Table Auto-generation Perl Code

Chapter 6

Prototype Design and Implementation

We have discussed the overall architecture of the CQ system and the underlying algorithms in the previous chapters. In this chapter, we will focus on the design and implementation details of the first prototype of the CQ system in the context of the *software life cycle*. Despite the difference among several software process models [31, 29], any software development is seen as an engineering process which follows the five basic steps: requirement analysis, specification, design, implementation (coding), and testing. In the following sections we report our design and implementation decision and effort with respect to these basic steps. The theoretical results obtained and shown in the previous chapters also serve as the baselines for the software development covered in this chapter. Thus we omit the technical feasibility study phase of software engineering and start from system requirements analysis. We also demonstrate, through this working prototype of the CQ system, the concepts of *continual query* and its usefulness for several mission-critical application domains.

6.1 System Requirements Analysis

System requirement analysis is very critical to the success of a software system project. The quality of requirement analysis depends to a great extent on how well and thoroughly the requirements for the target software have been analyzed. We proceed the requirements analysis following a top-down and general-to-particular process, in which software requirements are gradually refined and finalized in some form that is acceptable to the next phase, software design [31].

6.1.1 Analysis of Non-Functional Requirements

The non-functional requirements refer to the desirable properties of the prototype system. In this section we outline three most desired non-functional requirements that were considered at the initial prototype design stage: evolutionary prototyping approach, system and component portability, and system and component extensibility.

6.1.1.1 Evolutionary Prototyping

> *"A software prototype is similar to the idea of building a mock-up or model of something. It allows developers to understand complex problems by experiencing them first hand, and to use this experience to guide the development process". ([31])*

The CQ prototype system exploits the *continual query* concept, which itself needs testing and verification. At the beginning of the prototype implementation, we were facing several facts: first, the concept of continual queries is fresh, but the requirements and user needs have not been fully understood at the time when this project was initiated. Second, the functional and non-functional requirements of the CQ system still evolves as the need and the understanding of the CQ concept and expectation progresses. Therefore, the prototype must be flexible and extensible to allow seamless addition of new functionality. Thus, *Evolutionary Prototyping* [29] was chosen as one of the main non-functional requirement in the prototype building process. *Evolutionary Prototyping* is used to describe the system that has the capability to refine a solution given that the problem will evolve with time and that the system requirements evolve which in turn will generate new requirements.

6.1.1.2 Portability

One of the goals of the CQ prototype system is to create a value-added software package for event-driven update monitoring in an open distributed environment such as the Internet. This software prototype should be portable across platforms, and runnable by the applications via a network connection without a need for installation or compilation. All the necessary components must be downloadable and executable, possibly with the use of the necessary application viewing tool that is platform-specific but ubiquitous enough to be present at most platforms. For example, we need to provide a graphical user interface, which uses such standard interface components as control buttons, text components, execution logs, etc. Portability of such interfaces, at least for major platforms, should be achieved without having to implement separate versions of GUI for each platform.

6.1.1.3 Robustness and Extensibility

The proposed prototype implementation is an experiment – it covers neither all aspects of continual query processing and optimization algorithms nor all types of event composition operators. However, it must provide an extensible framework that would provide reasonable flexibility for adding new CQ optimization algorithms and new event composition operators as well as new mechanisms for event detection and change notification. The addition of new functionalities and algorithms should not affect the existing operation of the system nor the use of the existing functional components. Another desirable property is to allow these new modules be downloaded dynamically at run time, without the need to recompile the entire software package.

6.1.2 Analysis of Functional System Requirements

Functional requirements analysis is one of the most important steps in a software design process. In the functional requirement analysis step, we need to identify user needs and system constraints, and define system boundary, i.e., what functionalities the system is going to offer.

6.1.2.1 User Needs

The Internet and the Web have linked hundreds of thousands of data sources together all over the world. We can view the Web as an evolving information universe. Users can search and find useful information and at the same time publish information in any way and at any time. The amount of information sharing is phenomenal. However, in order to monitor a particular information source or a particular type of information sources such as stock trade, users typically have to go to the particular site where the source data is published, fuse and compare the data to obtain the changes. When the sources change frequently, one has to either visit the site more frequently or write a specific monitoring program to watch the updates at the source. The continual query system is intended to provide convenient and generic update monitoring services that allow users to use installed continual queries to watch the changes at the sources and notify the users whenever the changes reach certain update thresholds. Thus the user need for such system is an user-friendly GUI which let users to install the continual queries whenever there is a need of update monitoring, to stop the installed continual query by explicit specification of termination condition and the use of delete function to remove the installed CQ before it is expired. Other user-end services include browsing or modifying installed continual queries, and walking through different types of data sources being monitored as well as system administration services which are designed to provide convenient interfaces for the CQ system developers to test and monitor the system operations.

In summary, the main goal of the CQ system is to provide an easy-to-use, cross-platform *personalized data update monitoring toolkit* for online users. The user can define what kind of information he wants, when he wants it, and how he wants it. The CQ system will deliver the information to the user electronically (e.g., email, fax, pager, etc.) based on the user defined query and delivering criteria.

6.1.2.2 System Constraints

To start the prototype development with the existing computing environment and system software tools, we made certain decisions on the choice of platforms, operating systems, DBMSs, and programming languages (PLs). We view the following decisions as the major constraints of the first prototype CQ system:

1. Client Side

- Graphical Web browsers which supports HTML3.0 or above and Javascript. Java1.1 support is preferred for future compatibility

  Many services of the CQ system incorporate client-side dynamic HTML scripting with Javascript for easier and more friendly user interface. Java1.1 compliant browsers are preferred for future system add-ons. We have tested on Netscape 3.0 and Navigator (part of the Communicator suite). Other browsers which claim to support HTML3.0 or above and Javascript should work as well.

2. Server Side

   - Unix operating system

     The CQ server was developed on top of the Unix system. It is supposed to run under many different variants of Unix platforms, such as SunOS, Solaris, AIX, Irix, HP-UX, or Linux.

   - Programming languages

     Perl (Practical Extraction and Report Language) interpreter is used for the CQ system development. The system has been tested under Perl5 as well as Perl4. Perl is publicly available from the Web.

   - Relational database or meta data management

     In order to manage the meta data in CQ system as well as the simulation of online RDB data sources, we used an Oracle relational DBMS as the backend repository of the prototype. We have tested on Oracle RDBMS 7.1.6, 7.3.3, and Oracle8. Other RDBMS, such as Sybase and Informix, which support database triggers would also work.

3. Protocols, API's, and interfaces

   - HTTP protocol

     The underlying information transporting vehicle used in the CQ system is HTTP (Hyper-Text Transfer Protocol). HTTP is the underlying protocol used by the World Wide Web, which has become the center of the Internet activities because of its easy accessibility via a URL (Universal Resource Locator). Currently, we are using HTTP1.0.

   - SMTP Protocol

     SMTP (Simple Mail Transfer Protocol) is the standard e-mail protocol on the Internet. The CQ system currently uses email as the update notification method.

   - Web server

     We created our own web server to provide different services by the CQ system. Apache and Java Web Server (JWS) can be used for setting up the web servers. Both of them are publicly downloadable:

Apache: http://www.apache.org/dist

JWS: http://java.sun.com/products/java-server/jws111.html

- Relational database access interface through CGI or JDBC

  Access to the meta database and simulated source database through the Web is via CGI(Common Gateway Interface). Three options are available: a) Oraperl for Oracle 7.x [1]. b) DBI+DBD+Perl5 for Oracle8 [2]. c) JDBC.

- LIBWWW-Perl

  Libwww-perl is a library of Perl packages/modules which provides a simple and consistent programming interface to the World Wide Web. It supports HTTP/1.0. It's also publicly available at:

  http://www.ics.uci.edu/pub/websoft/libwww-perl.

6.1.3 Functional Components

The top-level function and information flow in CQ system are shown in Figure 6-1. This diagram also shows major subsystems and components in CQ system. Each component is connected with other components through the control and/or data flow relationships. Figure 6-1 presents a sketch of the *Architecture Flow Diagrams* [31] of the CQ prototype design. We will discuss detailed functionality of each component and the interconnections between components in subsequent sections.

In the first CQ prototype implementation, we group software components into five categories according to their functionality: (1) user interface processing, (2) Input processing, (3) Continual query processing, (4) Output processing, and (5) Maintenance and self-testing. To present a road map to the readers, we will first give a brief overview of each of these categories and then entering into a more detailed discussion.

6.1.3.1 User Interface Processing (UIP)

The UIP component provides an easy-to-use and uniform GUI to the CQ users. It takes the user's requests, passes them to other system components, and displays answers back to the user. The core of UIP is the *Form Entry and Report Subsystem*. A more detailed task breakdown of the UIP component is given in in Section 6.2.1.

6.1.3.2 Input Processing (IP)

The IP component is responsible to feed inputs to the *Continual Query Processing components* for further processing. Inputs include user information, normal SQL-like queries issued by the

[1] Oraperl is a version of Perl which has been extended to manipulate Oracle databases

[2] DBI (Database Interface) is a database access Application Programming Interface (API) for the Perl Language. It is database independent. DBD (Database Driver) is vendor-specific database driver module for Perl

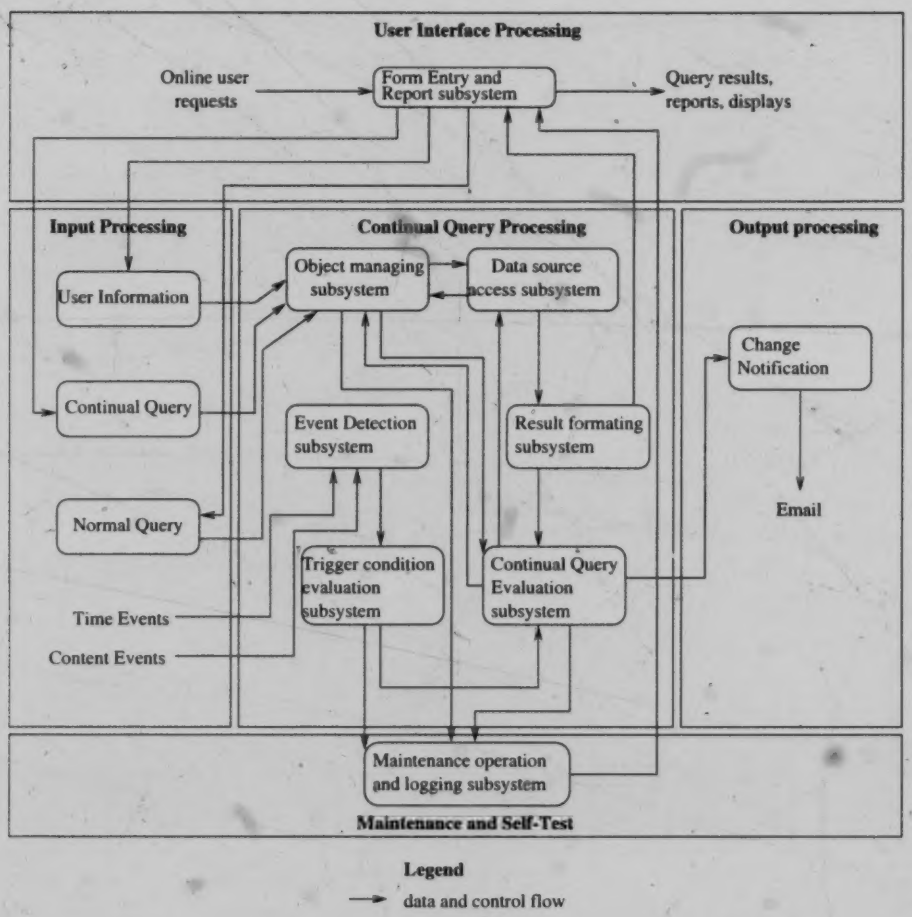


Figure 6-1: Top-level Architecture Flow Diagram of *Continual Query System*

CQ clients, continual queries installed by the CQ clients, including the event specification and the termination condition specification. Section 6.2.2 provides more discussion on the tasks of IP component.

6.1.3.3 Continual Query Processing(CQP)

CQP is the main component of the CQ system. It controls the actual processing of continual queries installed by the CQ users. CQP consists of six main subcomponents: object manager component, event detection manager component, condition evaluator component, continual query evaluation manager component, remote data access component, and result formatting component.

- The *object manager component* takes care of all the system and user objects.

- The *event detection component* detects all relevant events (time events or content-based events) and invokes the *trigger condition evaluation component* once certain events are detected and signaled the corresponding continual queries.
- The *trigger condition evaluation component* then evaluates whether the trigger conditions of the corresponding installed continual queries are true or false.
- The *continual query evaluation component* will be invoked if the trigger condition is true. The *continual query evaluation component* will then evaluate the query based on the naive (brute-force) algorithm or DRA. If a difference between the current and the previous evaluation of a specific continual query is detected, the *continual query evaluation component* will call the *change notification component* of the output processing subsystem.
- The *change notification component* of the output processing subsystem will inform the owner of the continual query about the new changes via email. The notification mechanisms may vary from email, to voice message, fax and other signaling methods.
- All the data source access is done via the *data source access component*.
- The results obtained from the data sources are processed in *result formatting component* before they are passed to other system components.

Section 6.2.3 provides detailed task specification of the CQP component.

6.1.3.4 Output Processing(OP)

The main purpose of the OP component is to send the change notification, including the specific continual query identifier and the differential result to to the creator or owner of the corresponding continual query. In the first prototype we plan to support the *change notification* via email only (see Section 6.2.4 for further details).

6.1.3.5 Maintenance and Self-Test(MST)

The MST component is designed to facilitate the testing of the prototype system. It includes *maintenance operation and logging component* which allow the user to test and diagnose other functional system components. Detailed description of the *maintenance operation and logging component* is given in Section 6.2.5.

6.2 System Design

In this section, we discuss the design of the main functional components of the CQ system shown in Figure 6-1. Our discussion proceeds in the order of user interface, input, continual query processing, output, maintenance and self-test.

6.2.1 UIP components

As discussed earlier, the UIP component provides the GUI interfaces to allow the CQ users to register the CQ system using User Registration Form module and query the CQ data sources without installation of the query as a continual query using the Normal Query Form. For any user who wants to install a continual query, he/she would need to log in using his/her userid and password. The installation of continual queries can be done by selecting time-based CQs or content-based CQs. Content-based continual queries may have triggering events that combine time event with content-based events. An example could be "Notify me whenever an airplane enters this sector for more than 5 minutes". The monitoring target event of this CQ is the sector and the constraint for this monitoring task is the time event 5 minutes. Once a user installs a number of continual queries, he/she may use the CQ browsing form to browse all the CQs he/she installed so far. Figure 6-2 shows the breakdown of the UIP components (connections to other CQ system components are not shown in this figure):

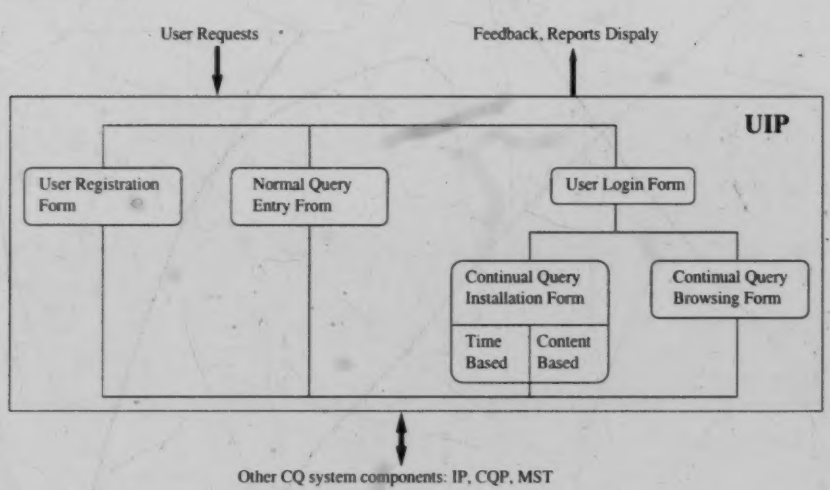


Figure 6-2: UIP components

6.2.1.1 Common GUI components

All the UIP components have two common GUI function buttons. one is the *Action Button* which carries out the functionality of the parent UIP component; [1] and another one is the *Reset Button* which sets the contents of the parent UIP component to original default ones. The *Action Button* may have different labels as the action function changes. Examples are "Submit Registration", "Submit Query", "Install CQ", or simply "Submit".

[1] We call component A the *parent* of component B if component A *contains* component B

6.2.1.2 User Registration Form

In order for user to install continual queries in the CQ system, the user has to register to the system first. The *User Registration Form* [2] is used to register relevant user information to the system. Once the user information is stored in the system, it is treated as a persistent object. The *User Registration Form* has the following GUI components, which are shown in Figure 6-7.

- *Login name input*

  This component is to enter the login name chosen by the user. It is used as the ID of a user. The maximum length for this component is 15 characters. Because every user is uniquely identified by his login name, we make sure there are no two users under the same user name. If the login name entered by the user has already been used by another user, the user will be prompted to try another login name. This component is mandatory.

- *Password input*

  This component is designed for the user to enter his password. For security considerations, every user is required to choose a password for himself. The password should be at least 4 characters and at most 12 characters long. The longer the password is, the more secure it becomes. But it will be more difficult to remember and type if the password is too long. The user will be asked for the password associated with his login name if he wants to install continual queries or get other CQ services later on. This component is required.

- *Password confirmation input*

  To insure that the user enters the password correctly, we require the user to confirm his password. This component is mandatory. If the contents in the *Password input* and the *Password confirmation input* mismatch, the user will be given an error message. He has to re-enter them.

- *Title input*

  This input component is to store the title or a particular user, such as Mr., Ms., Dr., Professor. This component is optional.

- *First name input*

  It is to take the input of the user's first name. This component is optional.

- *Last name input*

  Similar to the *First name input*, this component is to get the user's last name. It's also optional. But we recommend the user to input his first name and last name, thus he can receive more personalized emails from the system.

[2] Note that a *Form* here may physically be one Web page, span several Web pages, or be part of a page

- *Email address input*

  Because the system relies on emails to send users results of their installed continual queries, it is very important that users provide correct email addresses. This component is mandatory.

- *Subscription duration selection*

  The component is designed to record the length of the user's subscription. Once the subscription expires, the user is no longer a valid user in CQ system. But he can renew his subscription. The default subscription duration is one month. This component is mandatory.

6.2.1.3 Normal Query Entry Form

This function is designed for users who would like to pose ad-hoc queries to the CQ system rather than installing continual queries. Users can use *normal query* [1] to issue a query over the data sources accessible from the CQ system without being registered in the system. The *Normal Query Entry Form* has the following GUI components:

- *Data Source Selection*

  This component is to let the user to choose which domain of data sources he wants to query. In CQ, we build continual query services and wrappers according to different domains of interest. Examples include the logistics applications domain, the current weather watch domain, the bibliography services domain. Users may choose each domain by selecting the representative data source listed in this menu. The data sources could be in the form of relational databases, HTML web pages, bibtex files and other unstructured files. In this thesis, we will cover two types of application domains. One is the data sources wrapped using the *DIOM wrappers* and most of them are relational data sources, the other is *US Weather Watch* which wrap the data on US national weather service center web sites.

- *Table Description*

  This component is a hyperlink which, when clicking, will show the definitions of meta objects from the chosen domain of data sources. These definitions will be shown in HTML table format. This component is designed to help the user to browse the attributes and properties of source data objects.

- *SELECT clause input*

  This component is used for the entry of a list of output attribute names (or aggregate functions) of the query. The given list of attributes presents a projection list of objects which match the query condition specified in the WHERE input. This component is mandatory. By default, the selection is "*", which represents all the attributes from the object classes specified in the FROM clause of the query.

[1] We use SQL-like queries in CQ system simply because the query expressiveness of SQL(Structured Query Language) [10] and its wide use in relational databases. A SQL query is of the form `SELECT <attribute list> FROM <table list> WHERE <condition>`.

- *FROM clause input*

  This component holds a list of object class (table) names of the data sources, whose attributes may appear in the *SELECT* input. This FROM input is also mandatory. There are three sub-components in this component: a multiple selection list, a text box, and another display list. By default, all the available object class (table) names will be displayed in the multiple selection list. The user can choose one or more class (table) names from the selection list. For user convenience, the text box will show the total number of object classes (tables) selected in the selection list; and the display list will show the names of selected objects. Users cannot type in any input in the text box and the display list, but he/she can modify the selection in the multiple selection list.

- *WHERE clause input*

  This is a multiple line component where the user can type in the query condition. The condition could be a boolean expression which consists of several sub-conditions. The SQL syntax applies here. This WHERE class defines the selection condition of the query. When no query condition is given, the complete set of objects specified in the FROM clause will be returned with the projection list as the result format. This component is optional.

- *GROUP BY clause input*

  This component is used for the entry of grouping attributes, which appear in the *SELECT input*. The SQL syntax applies to *GROUP BY input*. An illustration of the usage of this field is given in the example below. The *GROUP BY* clause is optional.

- *ORDER BY clause input*

  This component specifies by which one or more attributes the query result is sorted. It is optional.

The layout of the *Normal Query Entry Form* is shown in Figure 6-9. An example usage of this form is

```
SELECT *
FROM ALP_INV_STORAGE, ALP_INV_ORDER
WHERE QTY_ON_HAND + QTY_ON_ORDER > THRESHOLD
GROUP BY ITEM_CAT
ORDER BY ITEM_NAME
```

6.2.1.4 User Login Form

For security reasons, we designed the *User Login Form* to check the user's ID and password before he can take certain actions. The two GUI components are:

- *User ID input*

- *Password input*

If a user login with an invalid user ID or password (e.g., the input user ID does not exist or the password is incorrect), an error message will be displayed. The *User Login Form* can be embedded in other UIP components.

6.2.1.5 Continual Query Installation Form

A registered user can install *continual queries* in the CQ system at any time. The *Continual query Installation Form* is the major GUI component in UIP.

There are two steps to install a continual query. Also, there are two types of continual queries: time-based and content-based. Time-based continual queries use time events are the trigger conditions and content-based continual queries use trigger conditions that combine time events with content-based events. For the first prototype implementation, we restrict that the content-based triggers contain no time events.

Each continual query installation process proceeds in two steps: login and fill-in continual query specification. We below explain the GUI components of each step:

Step 1: *login, select the domain of data sources and the type of continual queries.*

The *User Login Form* described in Section 6.2.1.4 is the GUI component used for user login service. Figure 6-11 shows a screenshot of the *User Login Form*. The *Data Source Selection* described in Section 6.2.1.3 is for data source selection and is also shown in Figure 6-11. The third GUI component in Step 1 is the *Trigger Type Radio Box input*. This component is designed for users to choose the type of the continual query that he wants to install, either time-based or content-based (recall the concepts in Chapter 3). The user may click either one of the two radio buttons to make the selection. Currently, time-based CQ can be installed for all the data sources wrapped by the CQ or DIOM wrappers. However we only support content-based continual query installation over *DIOM Data Sources*.

Step 2: *Enter query component, tirgger component, and termination condition component.*

The first component is the *Query Entry Form*. This component, as shown in Figure 6-12, is used to enter the query component of a continual query. The syntax description is the same as the query specification for *Normal Query Entry Form*, described in Section 6.2.1.3. The second component is the *Trigger Condition Entry Form*. Two types of trigger conditions (time-based and content-based) can be entered in this component. For the semantics and syntax of both types, see Chapter 3 and Appendix A for details.

For time-based trigger conditions, we support two types of temporal events in the CQ prototype system: (1) *absolute points in time*, defined by the system clock, e.g. "16:25, June 2, 1998". The precision is up to minute. (2) *regular or irregular time interval*, such as "every Monday" (regular) or "at 10:00 every first day of the month" (irregular) and "at midnight and noon every weekday" (irregular). The format of the time-based trigger condition is based on the specification

of UNIX crontab file [1]. The following GUI components are designed for users to enter their time events of interest:

- **Trigger Template Selection**

  This component is to provide some easy-to-use examples of trigger specification. Basically, after a user has selected a template from the selection list, this component will fill in the following five components respectively. It is designed primarily for the convenience and easy-to-use of users. However the templates covers only a subset of the fullest set of time event expressions that the CQ system supports. In order to fully exploit the expressiveness of the time-based trigger condition, we allow users to manually fill in the following five temporal components using the designated syntax.

- **Minute Input** can have values in the range 0 through 59.
- **Hour Input** can have values in the range 0 through 23.
- **Day of Month Input** has the domain range from 1 through 31.
- **Month Input** is in the range 1 through 12.
- **Day of Week Input** can have values in the range 0 through 6. Sunday is day 0 in this scheme.

Note that it is often not necessary to fill in all the five temporal fields together. But at least one of them should not be null. Otherwise, either a system default selection is chosen or there will be an error message displayed. To express a specific time specification, we may use a combination of the five components. Any of the five fields can be a list of values separated by commas. A value can either be a number, or a pair of numbers separated by a hyphen, indicating all the times in the specified range. The specification of days may be made by two fields: *Day of Month* and *Day of Week*. If both are specified as a list of elements, both apply. Some examples of time-based trigger condition is shown in the following table:

Min	Hour	Day of Month	Month	Day of Week
0	0	1,15		1-5
0	0,30			6,0

The first example sets the trigger condition to be at midnight every weekday as well as the first day and 15th day of each month. The second example says that the continual query trigger is set at every half hour on Saturday and Sunday. As we can see, this specification of time events is rather generic and powerful. We can use it to map almost all the natural language time expressions. Figure 6-13 shows the screen shot for this component.

[1] See crontab(5) in UNIX man page.

For content-based trigger condition, the basic event element (or basic event group) is expressed using the following expression:

$$Table.Attribute \ < ContentOp > \ [Value] \tag{6.1}$$

User may add extra components to the basic event group or combine several event groups together using event operators, namely to compose composite event groups and support richer expressiveness of the CQ trigger condition specification. The extra components are: *Group Functions, WHERE expression, GROUPBY expression*, and miscellaneous event operators [1]. In the CQ system, the trigger conditions are specified in the form of SQL-like expressions. As a matter of fact, the CQ triggers are evaluated as queries. Basically, each event group will be converted to an equivalent query expression over the data sources to detect the changes on the objects and their attributes of interest. The syntax for expressing the query equivalence of the basic event expression 6.1 is:

$$\texttt{select} < ListofAttribute > \texttt{from} < DataSourceNames > \tag{6.2}$$

The return values of the event query expression are compared with [Value] in the basic event expression 6.1 when evaluating the trigger condition. Further discussion on trigger condition evaluation will be continued in Section 6.2.3.3.

The content-based trigger condition installation has the following GUI components:

- **Group Function Selection**

  Let the user choose the group functions from the given list. The functions include: *AVG, COUNT, MAX, MIN, SUM*. When the user clicks the selection, the corresponding text will appear in the *Trigger Condition Multiple Text Input Board* described later.

- **Table Selection**

  This component when clicked will reset the contents of the *Attribute Selection* component to list only those attributes that belong to the clicked table object.

- **Attribute Selection**

  User can click an attribute whose value he wants to monitor. When clicked, this component will put the name of the attribute in the *Trigger Condition Multiple Text Input Board*.

- **Content Operator Selection**

  Currently, the content update monitoring operators supported in the CQ first prototype include: "CHANGES", "<>"(not equal), "="(equal), "<"(less than), ">" (greater than), "<="(less than or equal to), ">="(greater than or equal to), "CONTAINS"(substring), "LIKE"[2], "INCBY"(increase by value), "DECBY"(decrease by value), "INCBYP"(increase

[1] The syntax of the enriched trigger condition expression is given in Appendix A

[2] *LIKE* allows comparison conditions on only parts of a character string. It is similar to the "LIKE" comparison operator in standard SQL. Two wild card characters can be used in the comparison. '%' represents arbitrary number of characters and '_' replaces a single arbitrary character.

by percent), "DECBYP"(decrease by percent). These content operators can be classified in different ways:

1. *Unary or Binary operators*

   Only "CHANGES" is an unary operator. Other content operators are all binary operators.

2. *Numerical or String operators*

   We list the categorization of the operators in the following table.

Operator Type	Operators
Numerical	INCBY DECBY INCBYP DECBYP
String	CONTAINS LIKE
Hybrid	CHANGES <> = < > <= >=

3. *Cache or Non-cache operators*

   Some operators need to know the previous values of the attributes in order to perform the comparison operation. These are called *Cache operators*. Other operators only use the current value of the input attributes and thus are referred to as *Non-cache operators* (see the table below).

Operator Type	Operators
Non-cache	<> = < > <= >=
Cache	CHANGES INCBY DECBY INCBYP DECBYP

- **Value Input**

  This component takes the inputi string entered as the value of the attribute. When press `<Enter>` on the keyboard, the string value will display in the *Trigger Condition Multiple Text Input Board*. The value input can be either numerical value or character string. If it is a string, a single quotes (') is used to enclose it.

- **Grouping or Event Relational Operator Radio Buttons**

  Several sub-components are contained in this group of buttons: *WHERE Input, GROUPBY Input, Joint WHERE Operator Input, AND Input, and OR Input*. As those GUI components in the content-based trigger condition entry form, these subcomponents are either event composition operators or event constraint modifiers. They are used to specify the trigger condition of the continual query in the *Trigger Condition Multiple Text Input Board* when being clicked. We describe each operator as follows:

  - *WHERE Operator*

    Sometimes a single base event group is not sufficient to identify the real events of interest. For example, if we want to set the trigger condition to "when IBM's stock price is greater than $100", a simple event such as STOCK.PRICE ¿ 100 can not precisely express this condition. In this case, the base event of interest is the price of stock but our interest is only on IBM stock price changes, thus the base event `STOCK.PRICE > 100` has to be

specified with the constraint *STOCK.SYMBOL = 'IBM'*. In the actual implementation, we monitor the price changes and filter the IBM stock price as the conceptual event that we are interested in. That is why we use **WHERE** clause to explicitly distinguish the base event to be observed continually and the constraint that restricts the scope of the observation to those updates of interest. The following expression is the correct specification of the trigger condition "when IBM's stock price is greater than $100".

```
STOCK.PRICE > 500 WHERE STOCK.SYMBOL = 'IBM'
```

- *GROUPBY Operator*

  This operator is used to specify those events that contain aggregate functions such as the trigger condition: "notify me whenever the average sales of in each department are greater than $40,000". The condition expression is:

```
AVG(DEPT.SALE) > 40000 GROUPBY SALE.ITEM
```

- *Joint WHERE Operator*

  This operator is designed to combine different WHERE clause in the same event group. Basically, its function is the same as the boolean "AND". In order not to be confused with the reserved event logical operator "AND", we use "∧" to denote this operator.

- *Event Relational Operators*

  This group of operators are used to specify the relationships among all the event groups. Currently, we support "AND" and "OR" in the CQ system prototype. The *Truth Table* for the two operators is:

AND		
Event Group 1	Event Group 2	Value
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

OR		
Event Group 1	Event Group 2	Value
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Figure 6-3: Truth Tables for *Event Relational Operators*

- *Finish Radio Button*

  When the user completes the specification of the trigger condition, by pressing this button, his or her condition specification is committed (finalized) and the display of *Trigger Condition Multiple Text Input Board* is updated.

- **Clear Radio Button**

  This component is used to clear the content in *Trigger Condition Multiple Text Input Board* as well as other GUI components in *Trigger Condition Entry Form*.

- **Trigger Condition Multiple Text Input Board**

  This component acts as the display or input board of the current trigger condition string. User can use only mouse clicks (except the entry of attribute values). To input the trigger condition, users may selectively click some of the *trigger condition entry form*'s GUI components. The condition string in this board can also be updated using keyboard input as long as the user follows the syntax of the trigger condition.

Figure 6-15 displays the screen shot of the CQ content-based trigger specification.

The third component is the *Stop Condition Entry Form*. This component is used to specify the stop condition of a particular continual query. Currently, we only support an absolute stop time in the stop condition. This form consists of several selection lists and text input sub-components. They are: *Year Selection, Month Selection, Day Selection, Hour Input, Minute Input* [1]. Put differently, the time values are automatically set to the time zone in which the user is located. The default stop condition is two hours later than the current time. Of course, users can change their stop conditions as they wish.

After the user has completed the entries for *Query, Trigger Condition, and Stop Condition* and clicked the "Install CQ" action button, the input will be sent to the *Input Processing Subsystem* for verification. If there is no error in the input, this continual query will be installed at the CQ server. Otherwise, an error message will be displayed along with the diagnosis information.

6.2.1.6 Continual Query Browsing Form

This component provides the user a tool to view his installed continual queries or delete some of his/her installed continual queries if they are not yet expired but he/she would like to stop them. The GUI components are:

- User Login Form: It is the same as stated in Section 6.2.1.4, which provides access control to installed continual queries.

- Installed CQ Management Form: Using this component, a user can view and remove any or all the continual queries he/she has installed. The user can also input the ID of the continual query in the *CQID Input Box*, then press *Delete* to remove a single CQ from the system. Or he can choose *Delete ALL* to remove all the continual queries previously installed by him or her. However, a user can only delete his/her own installed continual queries. Otherwise, his action will be blocked and an error message will be displayed.

[1] Note that in a distributed environment as the Internet, times should be represented in a distributed manner, i.e., timezone information should be maintained along with the time specification. In the CQ system, we get the timezone information automatically from the Web browser and record this information. Thus the user doesn't have to input it.

Figure 6-16 and Figure 6-17 are the screens that users use to browse or delete his/her installed continual queries.

6.2.2 IP components

6.2.2.1 User Information

A *User Information* object is instantiated by the *User Registration Form* and will be passed to the *Object Manager subsystem* in *Continual Query Processing* unit. Then it is registered as a persistent object in the CQ system. It contains all the information about a valid user. The object description is as follows (the meanings of the attributes are self-explanatory):

```
CREATE Meta-Object User_Info {
    UserID              String,      //unique object ID
    Password            String,
    FirstName           String,
    LastName            String,
    Title               String,
    EmailAddress        String,
    Subscription        Date,
    SubscriptionDuration String
    }
```

6.2.2.2 Normal Query

The *Normal Query* object (See Section 6.2.1.3) is instantiated by the *Normal Query Entry Form* and passed directly to the *Data Source Access subsystem*. The result report is returned by the *Result Formatting subsystem* to the user through UIP components.

6.2.2.3 Continual Query Objects

This is another system maintained object as the *User Information* object. It is instantiated by the *Continual Query Installation Form* and passed to the *Object Managing subsystem*. It is then registered in the CQ system as a persistent object. It contains all the necessary information about a valid *continual query*. The object description is the following (the meanings of the attributes are self-explanatory):

```
CREATE Meta-Ojbect CQ_Info{
    CQID            long,       //unique object ID
    UserID          String,
    Name            String,
    Type            Char,
```

```
DataSources         String,
Query               String,
TriggerCondition    String,
StopCondition       DateTime,
Timezone            Char,
InstallDate         DateTime,
LastEvaluateDate    DateTime,
EvaluateNumber      Long
}
```

6.2.2.4 Events

There are two types of events that are monitored by the CQ system:

- *Time Event*

  Basic time events are initiated by clock signals in the CQ system. As mentioned earlier, the CQ system supports two types of time events: (1) absolute points in time(e.g., "11:59, December 31, 1999"); (2) regular or irregular time interval(e.g., "every hour" for *regular interval* and "midnight every first day of the month" for *irregular interval*).

- *Content Event*

  The basic content type events can be standard database operations such as INSERT, DELETE, UPDATE on single database objects. Furthermore, complex content events may be defined not only on individual data values or current database status, but also on sets of data objects and the transitions between database states. For example, a content event could be simply "IBM's stock price changes" or more complex: "IBM's stock price is increased by 10%".

6.2.3 CQP components

6.2.3.1 Object Manager Component

The *Object Manager Component* manages all the objects within the CQ system. Other CQP components have to coordinate with this component in order to access various CQ objects. The objects managed by this component include:

- *System Objects*

  System objects in the Continual Query System include: the *User Information* object, *Continual Query* object, and *Cache* object. The User Information object is discussed in section 6.2.2.1. The Continual Query object is discussed in section 6.2.2.3. Because *Continual Queries* are evaluated *continually*, we need to keep track of the "previous" results in order to give the user a report on new contents. Therefore, we keep the cache of the previous run (most recently) of the continual query for each user installed CQ.

- *User Objects*

  The user objects are objects that the client wants to monitor over data sources. For example, if the user has a time-based continual query installed in the CQ system: "tell me all the new Java books available for sale every Monday and stop by the midnight of December 31, 1998." It is expressed as follows:

```
Query:
    SELECT  title, author, isbn#, year, price
    FROM    BOOK
    WHERE   category='computer';
Trigger:
    every Monday;
Stop;
    12-31-1998 00:00:00
```

  In this example, the `BOOK` objects are user objects in the CQ object model, which has "title", "author", "isbn#", "year", "price" and "category" as its attributes (there are possibly other attributes for the `BOOK` objects).

6.2.3.2 Event Detection Components

We design two event detection components according to the two types of continual queries (recall Section 4.2.3 for concept definition and examples):

- *Time-based Event Detector*

  For time-based continual queries, a temporal event detector is used, which translates the time-based trigger condition into a clock event and registers the clock event to the CQ clock manager. The implementation of a clock manager is a system-specific decision. One may either choose to design a clock manager specifically for this purpose, or reuse the clock manager provided by an operating system. In the first prototype of the CQ system, it is implemented under Unix system, we make use of Cron as the clock manager because it is widely available under Unix-compatible platforms and is powerful for routine job scheduling. We can also make use of Cron's logging system to monitor and diagnose the execution. We define the semantics of the time-based trigger condition based on the *crontab* file format which is generic enough to express temporal events, no matter they are absolute time points, regular time intervals, or irregular time intervals.

  But there are some restrictions on the execution of Cron jobs. For example, when executing a Cron job, the Cron daemon usually starts up a Shell process to handle the job, which is not efficient. We are considering to write our own clock manager in the next prototype release to further enhance the efficiency of the system.

- *Content-based Event Detector*

  Given an installed continual query, the main task of the content-based event detector is to decide what to detect and how to detect.

  What to detect

  A *content-based continual query* is designed to monitor the user-defined events over the source data (compared with events over time). Therefore, the objects being detected or observed are not just time events as those cases in the time-based continual queries. The most common content-based events are updates on source data. Let us look at an example of content-based trigger condition (see Figure 6-15): "report to me *whenever IBM's stock price dropped by 5 percent or the maximum stock transaction volume is less than 2,390,000*". We can express this content-based trigger using the syntax described in Section 6.2.1.5 as follows:

  ```
  STOCK.price DECBYP 5 WHERE STOCK.companyname = 'IBM'
  OR MAX(STOCK.trans_volume) < 2390000
  ```

  Given the above trigger condition, we know that the events to be detected are **INSERT**, **DELETE**, and **UPDATE** operations on **STOCK.trans_volume** and **STOCK.price** because the updates on price are conducted by these operations and those changes may cause the trigger condition to change between **TRUE** and **FALSE**.

  How to detect

  Once we have identified **what to detect**, the next step is to decide **how to detect**. We need to choose the mechanism to detect the changes made by the update operations (**INSERT**, **DELETE**, and **UPDATE**). In the CQ system, we distinguish between the data sources that have built-in trigger capability such as the data sources managed by trigger-enabled RDBMSs(including Oracle, DB2, Sybase, and Informix) and the data sources that have no built-in trigger capability such as most of the Web sites and file systems.

  - For data sources with built-in trigger capability, the CQ system may install database triggers on data columns or objects of interests. Whenever there is an update on the trigger-monitored data objects, an update signal will be sent to corresponding CQ wrapper, which could include an update notification board that records which objects are updated and when they are updated. The information on the update notification board is open for external applications to query. We provide the host-specific trigger installation scripts (such as Oracle trigger installation Perl script) to install database triggers on data objects and data columns that are accessible to the CQ system.

– For data sources with no built-in trigger capability, we use CQ system controlled polling with system-defined interval (e.g., every 5 minutes).

6.2.3.3 Trigger Condition Evaluation Components

Trigger Condition Evaluation Components are designed to evaluate the trigger conditions of installed continual queries. Similar to event detection components, there are two *Trigger Condition Evaluation Components*:

- *Time-based Trigger Condition Evaluator*

  Time-based trigger condition evaluation is done by the clock manager, which is the Cron daemon in the currently prototype of CQ system. The clock manager checks the registered clock events in system-maintained event list (which is *crontab* file in our prototype). Once the clock event threshold is met, the clock manager triggers the *Continual Query Evaluation Component* to schedule a new round of CQ execution.

- *Content-based Trigger Condition Evaluator*

  A content-based trigger condition may be a composite event group composed of several basic event element which is a triplet of the form:

  ```
  <Table.Attribute> <ContentOp> <Value>
  ```

  The composite trigger condition construction operators are: *WHERE, GROUPBY, Joint WHERE Operator, and Event Relational Operators (AND/OR)* (Recall Section 6.2.1.5 for discussion on these operators).

  We use the example in Section 6.2.3.2 to illustrate how we evaluate a content-based trigger condition. In the example, the trigger condition is decomposed into two basic event groups:

  1. Group 1: `STOCK.price DECBYP 5 WHERE STOCK.companyname = 'IBM'`, which means "IBM's stock price dropped by 5 percent"
  2. Group 2: `MAX(STOCK.trans_volume) < 2390000`, which means the maximum transaction volume in market is less than 2,390,000.

  In addition, we identify the event relational operators among the event groups. In this case, it is the "OR" operator.

  When the *Content-based Trigger Condition Evaluation Component* is triggered by the content-based event detector, it is ready to schedule the trigger condition evaluation. First, it maps each individual event group to a source-specific query. If there are more than one event groups in the trigger condition, the *Content-based Trigger Condition Evaluation Component* will check to see (1) whether to evaluate the next event group and (2) which group to chose to be the

next based on the event relational operators. In the above example, event group 1 is evaluated first. If it returns FALSE, then the second event group will be evaluated. Otherwise, the whole trigger condition is TRUE without even evaluating event group 2. The truth table for *Event Relational Operators* are shown in Figure 6-3. Note that there are a number of ways that we can introduce optimization opportunity for condition evaluation. For instance, given a trigger condition expressed in the form of event groups connected with event operators, we can make use of the associativity to re-order the sequence of the event groups in a given trigger condition so that we can always choose to evaluate the *most inexpensive* (e.g., the event with high selectivity factor) event group first.

For the content-based trigger condition, we adopt standard relational operators as well as enhanced system defined operators as the *content operators*. They are: "CHANGES", "<>", "=", "<", ">", "<=", ">=", "CONTAINS", "LIKE", "INCBY", "DECBY", "INCBYP", "DECBYP". One thing to note is that these operators are not language specific. Instead, they are defined by the CQ system, independent of any specific platform. We can even override them. For example, in some programming languages, "=" is the assignment operator, while in CQ system, it is the relational operator that can be applied to either numerical values or string values. It is similar for "<>", "<", ">", "<=", and ">=". See Section 6.2.1.5 for more details.

6.2.3.4 Continual Query Evaluation Component

Once the trigger condition evaluation component evaluates the trigger condition to be true, then it signals the Continual Query Evaluation Component to fire a new iteration of the CQ query evaluation execution. The CQ *Query Evaluation Component* will first check to see if the installed continual query has expired. To do this, the CQ Evaluation Component talks to the *Object Manager Component* to get the *CQ_Info* object which records all the information about each installed continual query. The *current time* stamp is compared with the `Stop` condition. If the `Stop` condition satisfies, then this continual query will be removed from the CQ system (incl. Invoking the CQ garbage collector to clear all the cache generated by this continual query), and a notification message will be sent to the user. If the `Stop` condition of this CQ is not expired, the *CQ Query Evaluation Component* will compute the differential results by comparing the new query results with the previous cached results for a particular *continual query* and notify the user about new updates if there is any. The new results are then cached for the next iteration run of the continual query. Two CQ evaluation algorithms are Designed: naive and differential re-evaluation algorithms. In the first prototype only the naive algorithm is fully implemented for the CQ system prototype.

- Naive Algorithm

  The basic idea of *Naive Evaluation Algorithm* is to re-evaluate the Complete query over the raw data sources whenever the trigger condition is true. This algorithm is simple, but may

introduce a lot of unnecessary Computation as well as network traffic, especially when the update frequency of the data source is low, the amount of changes per update is small, the CQ trigger condition has short interval, and the number of continual queries installed over the same set of data sources is large (Recall Chapter 5 for detailed technical discussion).

- DRA

  Differential Re-evaluation Algorithm is designed to address the problems with *Naive Algorithm*. We have outlined the DRA algorithm, the benefit cases, and the implementation design detail in Chapter 5. Due to the fact that the implementation of DRA is still undergoing, we will not report more detailed implementation effort in this Chapter.

6.2.3.5 Data Source Access Component

The CQ system is designed to handle user queries over different types of data sources:

- Structured Data Source such as relational databases(e.g., Oracle, Sybase) and object-oriented databases(e.g., Objectstore, O2, Gemstone).
- Semi-Structured Data Source such as HTML pages and bibliography files.
- Un-Structured Data Source such as a LaTex file or C program files.

Each data source may have certain requirements or Restrictions (such as firewall) and its data presentation structures may not be Compatible with the object model of the CQ system. Therefore, the CQ system adapts the mediator-wrapper architecture and associate each data source (or each type of data source) with a wrapper to handle source-specific data access. The *Data Source Access Component* consists of a set of source wrappers. The source access is made transparent to external users (or programs). Figure 6-4 shows the context of the *Data Source Access Component*:

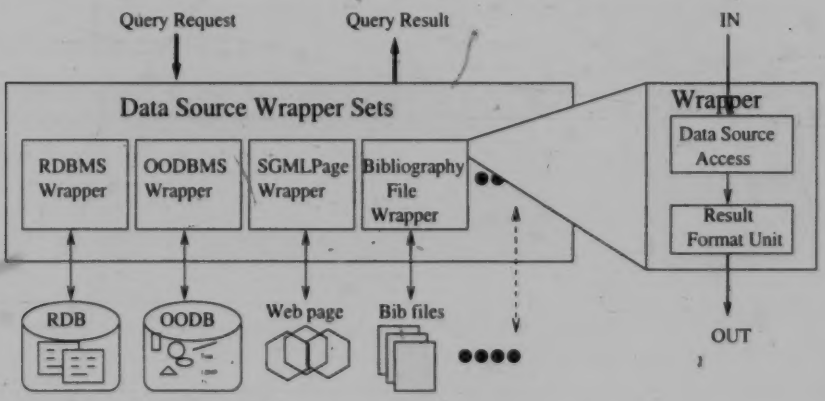


Figure 6-4: Wrapper Components

A relation database(Oracle) access wrapper, a weather web site HTML wrapper, a bibliography file wrapper are implemented in the CQ system prototype. We include a segment of Perl source code for Oracle database access in Section 6.3.

6.2.3.6 Result Formatting Component

After the query results are assembled from the multiple data sources, they have be formatted to the representation structure of the data consumer (such as the programs to display the results or postprocessing of the results). More concretely, if the results is passed to the user to be viewed in a Web browser, then the data consumer is the browser, and the display format is HTML; if they are to be passed to another CQ system component for processing, then the other component is the data consumer, and some special control information may be inserted into the raw data to meet the input requirements of the component. The context of *Result Formatting Component* is shown in Figure 6-4. In Section 6.3, we attach a fragment of the Perl routine source code which adds HTML tags to the raw data.

6.2.4 OP components

The component in the *Output Processing* subsystem is the *Change Notification Manager*. Its responsibility is to send change notification or status notification to the user whenever there is new result for his installed continual query returned by the continual query evaluator. We have designed a rich set of notification services to keep the CQ clients informed about their installed continual queries and to keep the other value-added CQ components informed of any abnormal situations.

Each notification policy needs to address the following questions:

- *When to notify*

  A notification will be sent out in the following situations: when a previously installed continual query expires; or a user installed continual query gets new result; or an error happened when evaluating the continual query; or the CQ system needs to be unavailable for some time.

- *Who to notify*

  The continual query expiration notification will be sent to the owner of the CQ; notifications about new query results are sent to the owner of the CQ as well; when an error occurred, depends on the type of the error, a notification will be sent to either the user(an application error, such as data query over the source times out) or the CQ system administrator(a system error, such as unable to create cache); when the CQ server is going to be upgraded or has to be down for some system maintenance reason, the server will broadcast a message to all registered users.

- *What to notify*

  An error message containing detailed error information(error type, time, context, etc.); a URL

where the user can find the diff-report of the new continual query evaluation result; or a broadcast message about some incidences happened on the CQ server.

- *How to notify*

  Emails are used as the notification method in the current prototype. We are looking into the possibilities of incorporating other notification methods such as fax, voice mails into the CQ system.

A sample notification upon detecting new results of an installed continual query is as follows(continual query #208 by user "wtang"):

```
Subject: Your installed CQ #208 has new results!
To:      wtang@cs.ualberta.ca

Dear Mr. Wei Tang,

Your registered Continual Query #208 got new contents.
You can find it at:
    http://chinook.cse.ogi.edu:8888/~cq/CQresult/CQwtang208.html

For more information on the CQ project, please visit us at:
    http://www.cse.ogi.edu/DISC/CQ

To play with the CQ prototype demo, please visit:
    http://chinook.cse.ogi.edu:8080/~wtang/CQ/html/CQ.html
or:
    http://www.cse.ogi.edu/DISC/CQ/demo

Best regards,
The CQ project team (cq@cse.ogi.edu)
```

The corresponding result page (CQwtang208.html) is shown in Figure 6-5.

6.2.5 MST components

In order to facilitate the debugging and maintenance effort of the CQ system, we keep diagnostic and log information of major continual query execution steps. An example is the trace log we generate for each CQ installation execution As shown in Figure 6-14.

Other MST components include CQ system administration services which allow authorized users to experiment the updates at the data source(simulated) and watch how the CQ system evaluates

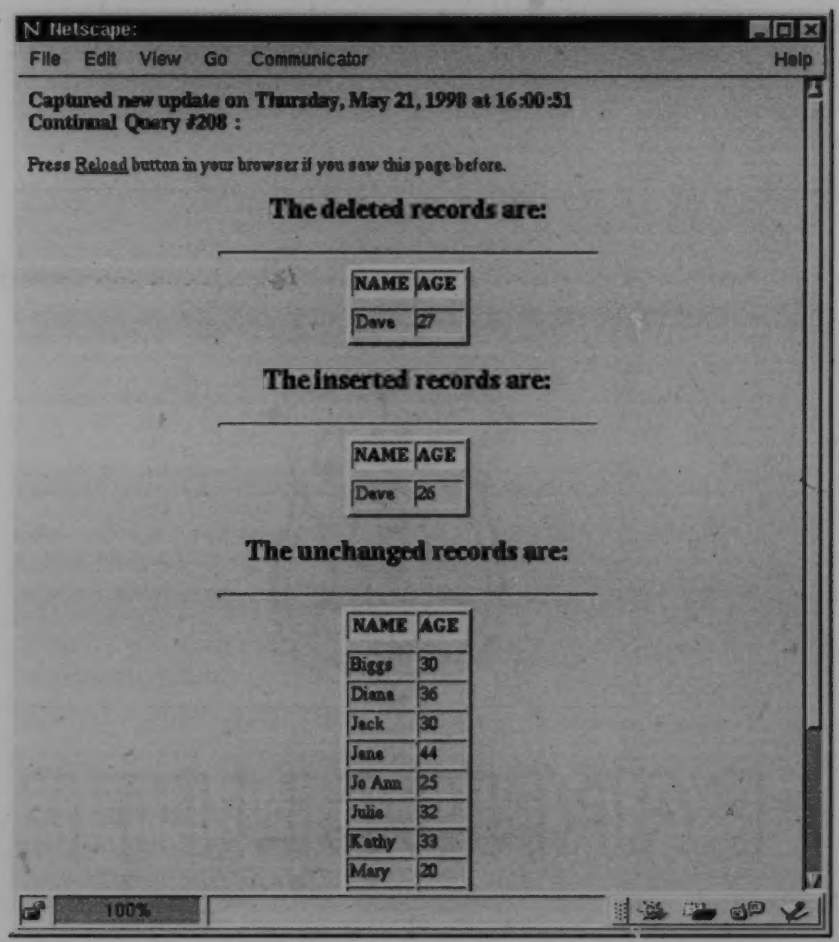


Figure 6-5: Sample Continual Query Result Page

the trigger condition, get the notification message, and checks the new query results. We also have facilities for users to view installed or delete continual queries. What is more, we can manage all the meta data through a simple easy-to-use web CGI interface and view the meta data of the CQ system in a well-structured HTML table format which is much better and more convenient than using the standard database utilities such as SQL*Plus. It is fast as well. By doing this, we can closely observe the behavior of the CQ system and debug the system more efficiently.

6.3 Coding Design

In the current CQ system prototype, we use Perl script as the server-side processing language. We also use Perl CGI (Common Gateway Interface) scripts and JavaScript to handle HTML Forms processing and provide enhanced dynamic HTML pages.

6.3.1 Perl Programming Language

Perl (Practical Extraction Report Language) is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It provides powerful regular expressions in pattern matching, and can be used to easily search or replace patterns in strings or text files. In the CQ system, the data we process most are HTML web pages, query results from the database or cached text files. Therefore, we take advantage of Perl's power in text scanning and manipulating in building the prototype system.

Furthermore, Perl is an interpreted language. Thus we do not need to compile it every time we run it. It is also quite easy to debug Perl scripts. Of course, since Perl is interpreted, it may not run as fast as compiled programs. However, in the CQ implementation, we do not use Perl to perform CPU intensive jobs. All the time-consuming tasks such as database query handling are done by a powerful database server. Perl is used simply to parse the query result and manipulate the texts. After test-running of the CQ prototype system for months, we have found that Perl is pretty capable for the work. It would be interested to try out other language options, such as Java and C, through a thorough performance analysis and evaluation.

6.3.2 Common Gateway Interface

The Common Gateway Interface (CGI) is a standard for external gateway Programs (CGI programs) to interface with information servers such as HTTP servers or Web servers.

A CGI program is any program designed to accept and return data that conforms to the CGI specification (see `http://hoohoo.ncsa.uiuc.edu/cgi/interface.html` for further details about CGI specification). CGI is quite handy in processing HTML FORMs on the Web. The following is a piece of HTML source code with a FORM in it:

```
<HTML>
<TITLE>FORM Example</TITLE>
<BODY>
<FORM method=GET
   action="http://chinook.cse.ogi.edu:8080/~wtang/cgi-bin/formtest.cgi">
<INPUT TYPE=text NAME=test>
<INPUT TYPE=submit>
</FORM>
</BODY>
</HTML>
```

Basically, the Web browser will pass the form input to the Web server (called chinook.cse.ogi.edu, 8080 is the port number) as an HTTP GET request. The server then starts a process to run the CGI program and passes back to the client the output from the CGI program (`formtest.cgi` in

this case). The CGI program can be written in any programming language, including C, Perl, Java, Visual Basic or even Unix shell scripts. Take my **formtest.cgi** as an example:

- In Perl

```
#!/usr/local/bin/perl
##########################################
#get the input from an HTML form
#then display its content in another page
##########################################
print "Content-type: text/html \n\n";
print "The input is: $ENV{'QUERY_STRING'}";
```

- In Shell script

```
#!/bin/sh
echo "Content-type: text/html"
echo
echo "The input is: $QUERY_STRING"
```

These two scripts simply display in an HTML page the input from the web form and print out its content. We use these simple examples to show how a CGI script interacts with the web client and the server.

6.3.3 Dynamic HTML

In contrast with *static HTML* which means an HTML page that never changes, *dynamic HTML* refers to HTML Web pages that may change each time it is viewed. For example, the same URL could result in a different page depending on the number of parameters used, such as:

- Geographic location of the reader
- Current time
- Previous page viewed by the reader
- Profile of the reader

There are many technologies for producing dynamic HTML, including CGI scripts, Server-Side Includes (SSI), cookies, JavaScript, and ActiveX. We choose CGI scripts and JavaScript to generate dynamic HTML pages. Compared with Java, JavaScript is more light-weight and is well supported by various Web browsers [1]. JavaScript does not have the programming overhead of Java. We do not have to wait for minutes for the JavaScript program to be initiated as Java program does, though JavaScript programs are less powerful than Java programs. Unless we want to process a large set of objects and intensive user interactions, JavaScript is more light-weight and easier to use. In the

[1] JavaScript was developed by Netscape which can interact with HTML source code, enabling Web authors to spice up web pages with dynamic content. It is endorsed by a number of software companies and is an open language. Microsoft Internet Explorer also supports a large subset of JavaScript, which it calls Jscript.

CQ prototype design, we do not have a lot of GUI components and user interactions. Much of the processing is done offline. Another advantage of combining client-side JavaScripts and server-side CGI scripts is that we can balance the workload between the client-side and the server-side by moving some of the error checking and data manipulation from server-side CGI scripts to client-side JavaScripts. Also we usually pass the user requests in a chunk to be processed by the server-side CGI scripts in order to reduce the communication cost.

Below we show a fragment of the CGI script that generates the HTML Representation for the raw query results returned from an Oracle SQL query.

```
#############################################################
# In:   ($dbname, $login, $passwd,$query)
# Out:  $csr        #raw query result(an Oracle cursor)
# call: print &runQuery($dbname, $login, $passwd,$query);
#############################################################
sub runQuery{
  local($dbname, $login, $passwd, $query) = @_;
  local($lda, $csr);

  &setOraEnv($dbname, $login, $passwd);
  $lda = &ora_login($in{"DB"},$in{"LOGIN"},$in{"PASSWD"})
        || &oracleError;
  $csr = &ora_open($lda, $query) || &oracleError;
  return $csr;
}

#############################################################
# set global environment variables
# In: $dbname, $login, $passwd
#############################################################
sub setOraEnv{
  local($dbname, $login, $passwd) = @_;
  $in{"DBMS"} = "ORACLE";
  $in{"DB"} = $dbname;
  $in{"LOGIN"} = $login;
  $in{"PASSWD"} = $passwd;
}

#############################################################
# print out Oralce error message
# Out: exit the program
#############################################################
sub oracleError{
  print "<H1>Oracle Error</H1> <HR>";
  print "$ora_errstr";
  exit(1);
}
```

6.3.4 Online Database Accessing

CGI scripts have been the initial mechanism used to make Web sites interact with databases and other programs, especially Perl CGI scripts. The latest Perl5 provides better support for remote database accessing. It has a built-in DBI(DataBase Interface) module which coordinates with other vender-provided DBD(DataBase Driver) modules for various database accessing. In the CQ proto-

type system, we tried initially using Oraperl (an extended Perl4 with database access capabilities) to access Oracle 7.1.6 database. During the development of the CQ system, we experienced database upgrading (from Oracle7 to Oracle8). We then moved to Perl5 with DBI and DBD modules. The transitions were seamless, which prove the great portability of Perl scripts and the database access modules.

There are other promising alternatives to the Perl DBD (DBI) CGI accessing of online databases. Java JDBC (Java DataBase Connectivity) has drawn great attention of the Internet community. JDBC is a programming interface that allows Java applications to access databases via the SQL language. Since nearly all relational database management systems support SQL, and because Java itself runs on various platforms, JDBC makes it possible to write a single application that can run on different platforms and access different DBMSs.

Below we use a fragment of the CGI script for remote access to Oracle databases as an example to illustrate the code design of this functionality.

```
#########################################################################
# In:   $csr     #raw result of the query(an Oracle cursor)
# Out:  $table   #an HTML table string
# call: print &runQueryHTML($csr);
#########################################################################
sub runQueryHTML{
  local($csr) = @_;
  local($table, @colstitle, $numOfCols, $rownum, @rowdata);

  @colstitle = &ora_titles ($csr, 0);        #get column names
  $numOfCols = $#colstitle + 1;              #get total column number
  $rownum = 1;

  $table = "<table border=2>";

  # making table headers
  $table .= "<th bgcolor=cyan align=middle >#</th>";
  for ($i=0; $i < $numOfCols; $i++) {
    $table .= "<th bgcolor=yellow>";
    $table .= "$colstitle[$i]";
    $table .= "</th>";
  }

  #making table contents
  while (@rowdata = &ora_fetch ($csr)) {      #get table row data
    $table .= "<tr>";
    $table .= "<td bgcolor=cyan> $rownum </td>";
    for ($i = 0; $i < $numOfCols; $i++) {
      $table .= "<td>";
      if ($rowdata[$i] !~ /^\s*http:\/\//) {
        #if the column text is not a URL
        $table .= "$rowdata[$i]";
      }else{
        #if the column text is a URL, the add HTML anchor tag
        $table .= "<a href=$rowdata[$i] > $rowdata[$i]</a>";
        $table .= "</td>";

#recognize image type and add a thumbnail image
```

```
        $suffix3 = &upper(substr($rowdata[$i], -3, 3));
        $suffix4 = &upper(substr($rowdata[$i], -4, 4));
        if ($suffix3 eq "GIF" ||                          #GIF  image?
            $suffix3 eq "JPG" || $suffix4 eq "JPEG")      #JPEG image?
        {
          $table .= "<td>";
          $table .= "<a href=$rowdata[$i]><img src=$rowdata[$i]
                    width=40 height=40></a>";
          $table .= "</td>";
        }
      }
      $table .= "</tr>";
      $rownum++;
    }
  }

  $table .= "</table>";
  return $table;
}
```

6.3.5 Emerging Technologies

Recently, Java servlets are becoming increasingly popular as an alternative to CGI programs. A Java applet runs in a Web browser environment, whereas a Java servlet is a piece of Java code that runs on a server.

The biggest difference between Java servlets and CGI programs is that servlets are persistent, which means that once a servlet is started, it stays in memory and thus can fulfill multiple requests. In contrast, a CGI program disappears after it serves a request. When next time another request comes, the server has to start a totally new CGI program process to answer the request. A lot of overhead is introduced in this way, especially when user requests are frequent and large.

In order for Java servlets to function, a Java Virtual Machine (JVM) must run on the server, and the server has to support the Java Servlet API as well. JVM is widely available for many platforms. Java Servlet API is a set of Java classes that can be downloaded directly from Sun. It also comes with a set of simple plug-ins that add servlet support to Web servers like Netscape, IIS (Internet Information Server *from Microsoft*) and NCSA's Apache server. Java Web Server(JWS) come with built-in servlet support.

Servlets are very promising for sharing information among multiple invocations of the servlet and among multiple users because servlets are persistent. For example, if a user want to submit something to a database, he can start the task in separate Java threads and they are working in parallel.

Another good feature of servlets is that they are modular, each servlet can perform a specific task and they can talk to each other when tied together via "servlet chaining", which means the output of one servlet can be forwarded to the next servlet in a "chain" to process.

We have begun to convert some the CGI programs to Java Servlets. We use Java Web Server and JDK1.1.5, servlet development and configuration are both quite smooth. We use JDBC in Java

servlets to handle database access. We are also planning to use Java programs simply because it can provide more GUI components and support better user interactions. The only problem is that it may takes quite some time for the browser to load the applets the first time. We believe this can be solved in the near future with the new generation of Java technology.

Java at client side(applet) provides rich GUI components and can handle user Interactions handsomely, while Java at server side(servlet) speeds up performance. Thus we make good use of Java's "write once, run anywhere" feature and maximize the system's portability and extensibility. This combination has a bright future. Client-side JavaScript is also a good candidate because it is more light-weighted than Java applet.

6.3.6 Online Live Demo of the Prototype

The online demo can be accessed using a JavaScript-enabled Web browser through the following URL:

`http://chinook.cse.ogi.edu:8080/~wtang/CQ/html/CQ.html`

6.4 User Interface Walkthrough

In this section, we walk through the user interface usage of the CQ prototype system, which is called "Continual Query Services". The content of this section is targeted at the end-users of the software package.

6.4.1 Main Menu Window

The top level page has a frameset which displays a CQ banner at the top, a service content menubar at the bottom which has the links to all the CQ services, and the main display area in the middle:

6.4.2 Client Services

6.4.2.1 User Registration

To become a valid user in the CQ system, one has to register to the CQ system using *User Registration Form* which is shown in Figure 6-7. The main controls are:

Choose a Login Name text box allows the user to input a unique login name.

Choose Password text box is to let the user type his password. A good password usually has more than 4 characters and less than 12 characters.

Confirm Password field is for the user to re-type his password to ensure the password he entered in the *Choose Password* box is what he expected.

Title input box holds the string of the title of the user. It is optional.

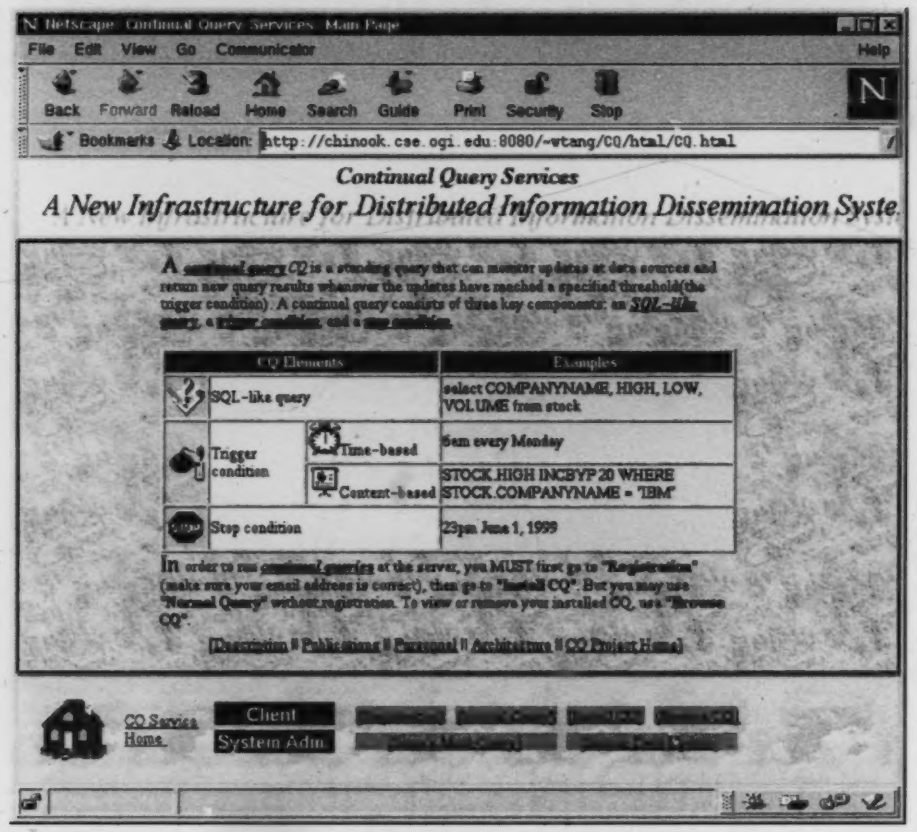


Figure 6-6: The CQ Services Main Window

First Name allows the user to fill in his first name. This field is optional.

Last Name is for the user's last name input. It is also optional.

Email Address text input box is used for the user to type in his email address where he will receive messages from the CQ system. This field **MUST** be filled in with the user's correct email address.

Subscription Duration selection control has the value list for the length of the user's subscription to CQ system. The default length is one month. If the user's subscription expires, he will be removed from the CQ system.

Submit Registration button submits the current content of the page to CQ system. Upon success of the subscription, the user will get a confirmation email.

Reset button clears all text fields on the page and set the *Subscription Duration* to the default value.

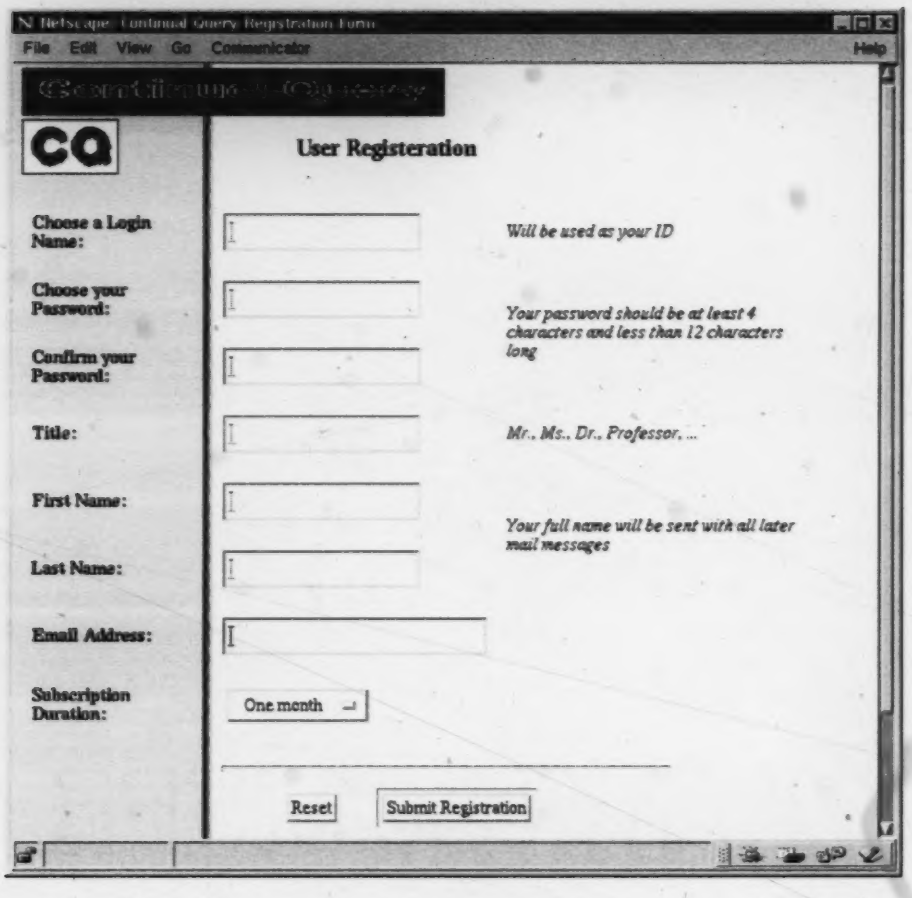


Figure 6-7: User Registration Form

6.4.2.2 Normal Query

Before a user subscribes to CQ system, he may use *Normal Query* service to query over the data sources wrapped by the CQ system and get a flavor of what kind of data are accessible. In order to pose a normal query, a user needs to choose the type of the data sources (i.e., a specific domain of interest) first, as shown in Figure 6-8.

After the user chooses DIOM Data Sources from the selection list and clicks the Submit button, he will be brought to the screen shown in Figure 6-9 to construct his query component of the CQ. The main controls are as follows:

See Table Descriptions hypertext is linked to a form which describes the types of the attributes for all the objects(tables) at the data source. This is helpful for the user to refine his queries.

SELECT , FROM, WHERE, Group By and Order By are used to construct the SQL query. The *FROM* selection list is automatically filled in with all the data objects(tables) available at the data

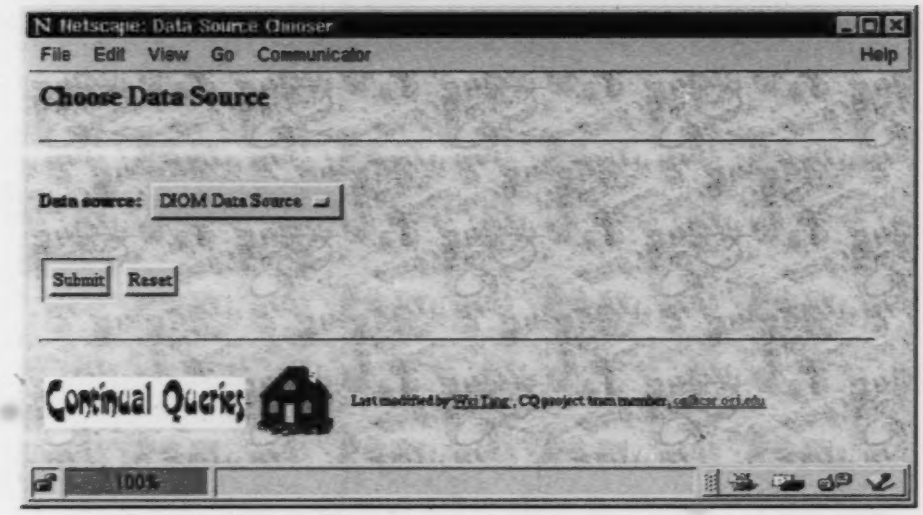


Figure 6-8: Choose Data Source Screen

source. By default, the query will return all the attributes of the objects selected in *FROM*, which is denoted by a '*' in the *SELECT* text input box.

Selected text box displays the current number of objects the user has selected. It will change according to the user input in *FROM* selection list.

Table List display box lists the names of selected data objects.

Reset button sets the values for all inputs to default.

Submit Query button submits the query.

When the user clicks the *Submit Query* button, a new page with the execution log information and the query result will be displayed. A sample query result page is shown in Figure 6-10 (the query is *select * from IMAGEGALLERY*):

6.4.2.3 Installation of a CQ

The first step to install a continual query in the CQ system is shown in Figure 6-11.

User ID and Password text inputs are used to type in login information. They are used to check if a user is a registered user. If not, an error message will come up.

Data Source selection list allows the user to choose the data source from a list to install his continual query on. There are currently two data sources: *DIOM Data Source* and *US Weather Watch*.

Trigger Type radio button group is to let the user choose the type of the continual query between time-based and content-based. The default type is time-based.

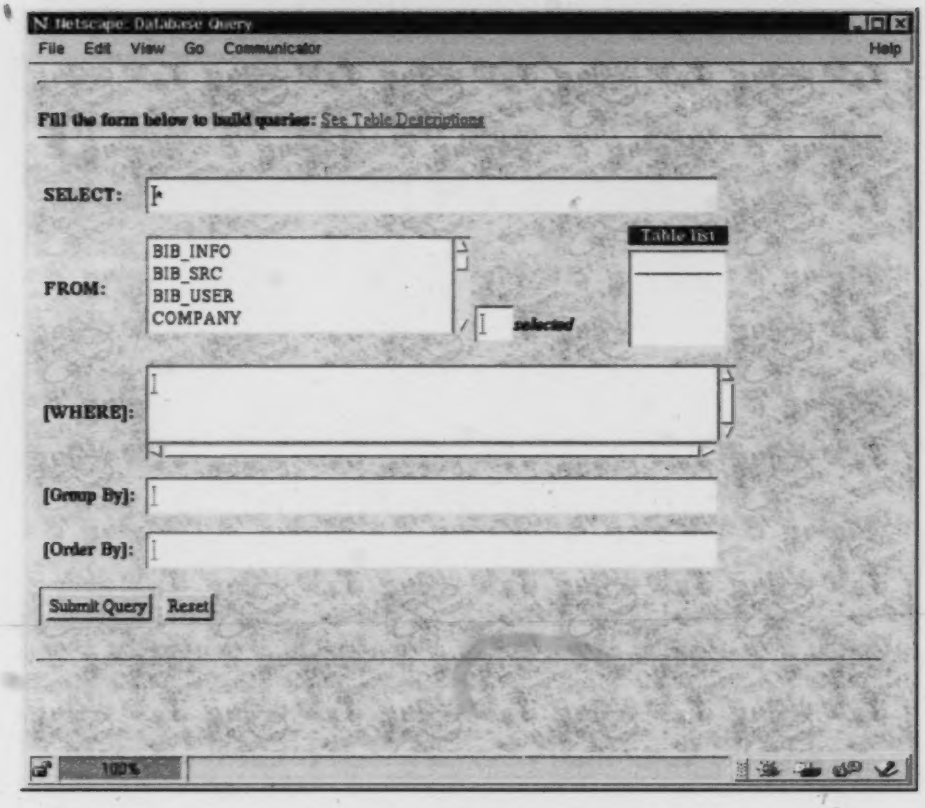


Figure 6-9: Normal Query Entry Form Screen

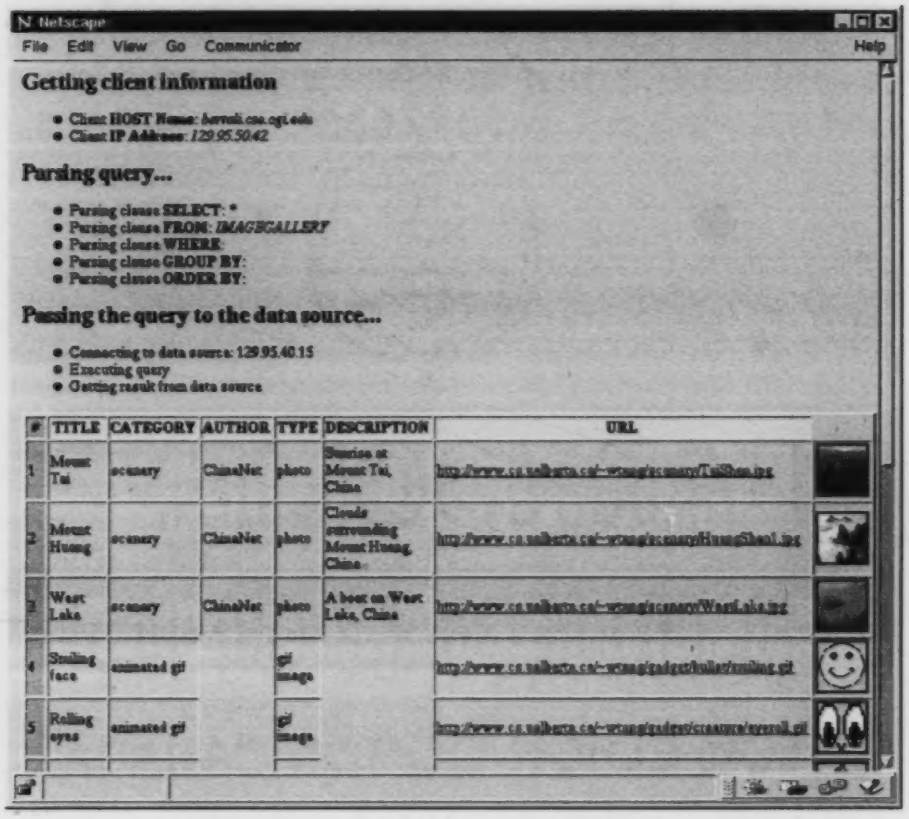

Netscape
File Edit View Go Communicator Help
Getting client information
Client HOST Name: kerouk.cse.cqi.edu
Client IP Address: 129.95.50.42
Parsing query...
Parsing clause SELECT: *
Parsing clause FROM: IMAGEGALLERY
Parsing clause WHERE:
Parsing clause GROUP BY:
Parsing clause ORDER BY:
Passing the query to the data source...
Connecting to data source: 129.95.40.15
Executing query
Getting result from data source
TITLE CATEGORY AUTHOR TYPE DESCRIPTION URL
1 Mount Tai scenary ChinaNet photo Sunrise at Mount Tai, China
2 Mount Huang scenary ChinaNet photo Clouds surrounding Mount Huang, China
3 West Lake scenary ChinaNet photo A boat on West Lake, China
4 Smiling face animated gif gif image
5 Rolling eyes animated gif gif image

Figure 6-10: A sample result page for Normal Query

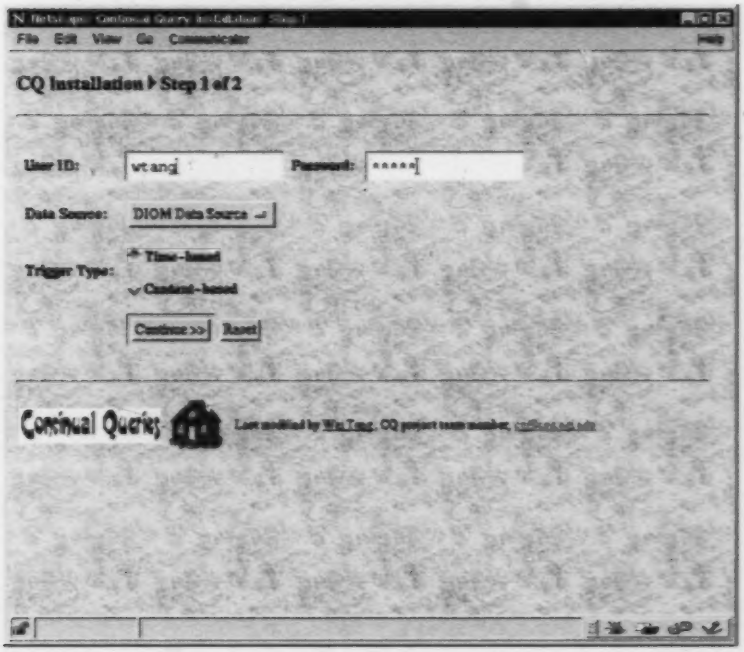


Figure 6-11: CQ Installation Step 1 screen

Continue >> button is to enable next step of the installation.

Reset button will clear all contents in the text fields and set all other controls to default values.

In the second step, the user will fill in the information for the continual query he wants to install, which includes *Query*, *Trigger Condition* and *Stop Condition*. Based on the type of the trigger condition, the interfaces are slightly different.

- Time-based Installation in Step 2

  The interface for the query construction part is shown in Figure 6-12. The controls are:

  CQ Name text input allows the user to type in a name for his continual query. It can act as a reminder of the content of the CQ. This field is optional.

  SELECT , FROM, WHERE, Group By, Order By, selected and Table List are the same as those in the *Normal Query Form* which is displayed in Figure 6-9.

  The interface for the trigger condition and stop condition installation is presented in Figure 6-13. The controls are:

  Templates selection list holds a list of time-based trigger condition templates which are more frequently used by users. On selecting any of the choices in the list, corresponding text fields below are filled with appropriate strings.

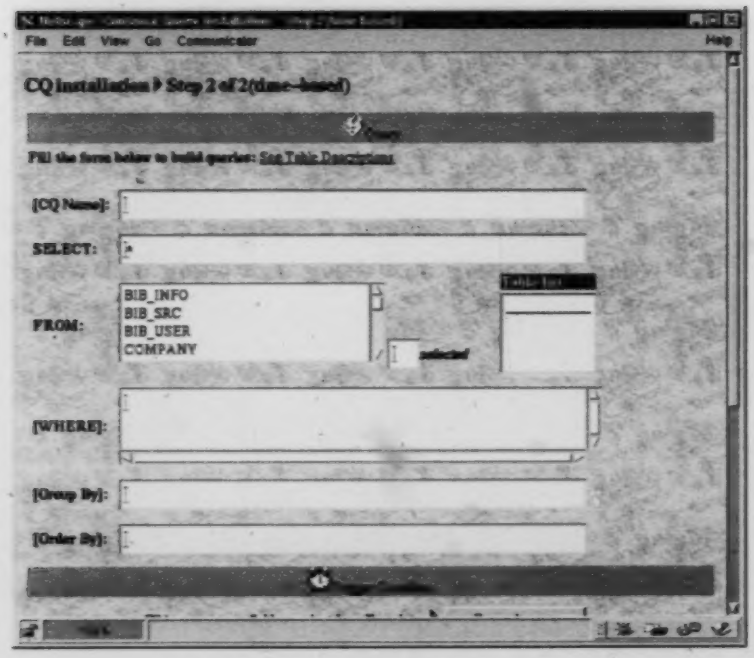


Figure 6-12: CQ Installation Step 2 (time-based) - Query Construction

Min text input box lets the user type in minute specification in the trigger condition.

Hour text input box lets the user type in hour specification in the trigger condition.

Day of Month text input box is for the input of *Day of Month* specification in the trigger condition.

Month text input box lets the user fill in month specification in the trigger condition.

Day of Week text input box is to let the user input *Day of Week* information in the trigger condition.

Stop Year selection is to choose the *stop year* for the continual query.

Stop Month selection lets the user choose the *stop month* from the list.

Stop Day selection specifies the *stop day* for the continual query.

Stop Hour and **Stop Minute** are two text fields which allows the user to input the hour and minute information for the stop condition specification.

Install CQ button will submit the contents of the page and pass them to the CQ system to install the continual query. Upon success, the user will see a page similar to Figure 6-14.

Reset button clears the contents of all the input components or set their values to default.

Help button when clicked will bring up a new page with help information of how to install a time-based CQ

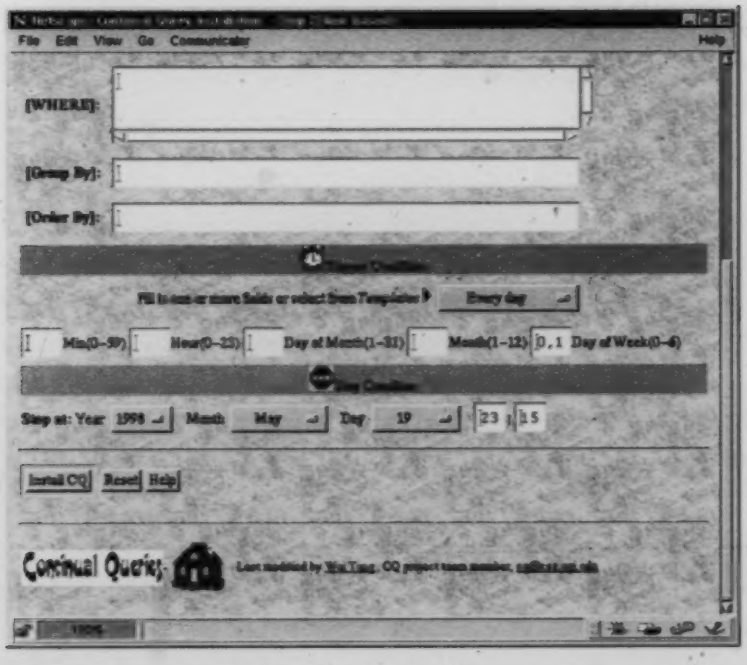


Figure 6-13: CQ Installation Step 2 (time-based) - Trigger and Stop Condition

- Content-based Installation in Step 2

  The interface of the query construction part for content-based continual query installation is similar to that for time-based installation(except the page title) as shown in Figure 6-12.

  The interface for the content-based trigger and stop condition installation is presented in Figure 6-15. The descriptions of the controls are as follows:

  - *For the trigger condition input*, the GUI components are described below:

    Group Function Selection, Table Selection, Attribute Selection, Content Operator Selection, Value Input, Grouping or Event Relational Operator Radio Buttons (WHERE, GROUPBY, Joint WHERE Op), Event Relational Operators (AND, OR) controls
    are all visual controls which help the user enter the content-based trigger in the Trigger Condition Multiple Text Input Board. When they are clicked, their associated values will be flushed to the Trigger Condition Multiple Text Input Board whose content represent the current value of the trigger condition. The user may also type the trigger condition directly in the Trigger Condition Multiple Text Input Board provided that he follows the syntax. (See Section 6.2.1.5 for details)

    Finish button
    is to update the current value of the trigger condition to the Trigger Condition Multiple Text Input Board and finalize the condition string.

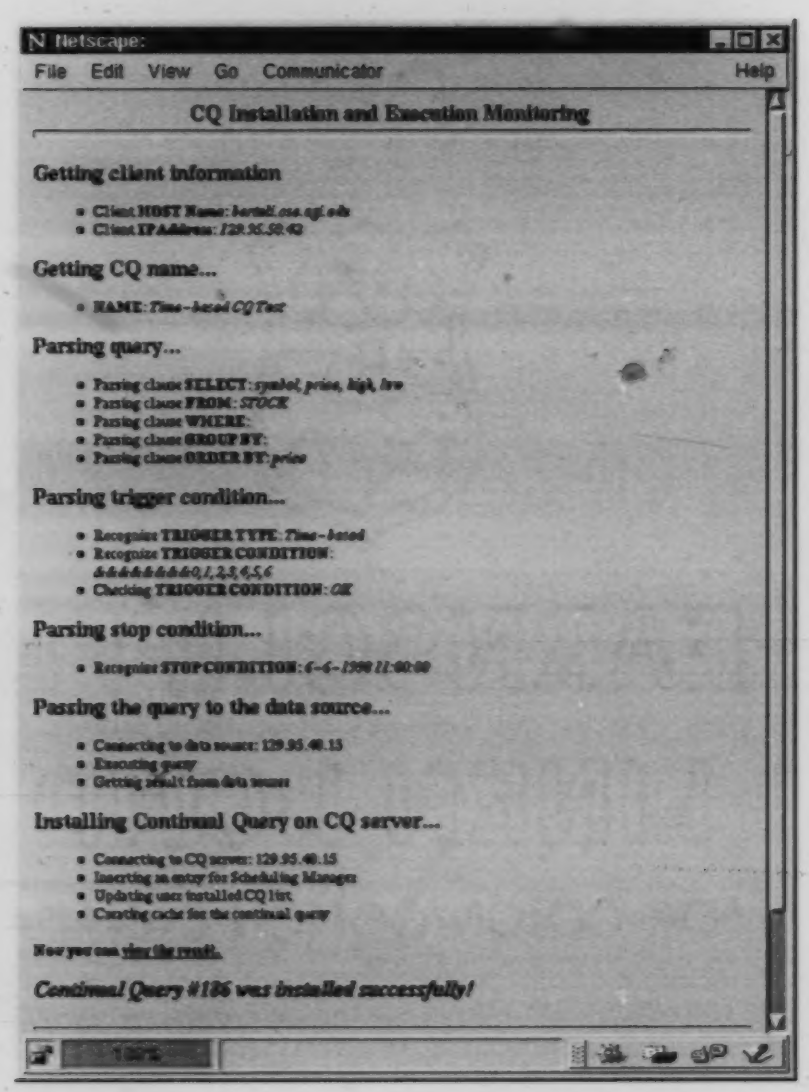

Netscape:
File Edit View Go Communicator Help
CQ Installation and Execution Monitoring
Getting client information
Client HOST Name: bertil.cse.ogi.edu
Client IP Address: 129.95.50.42
Getting CQ name...
NAME: Time-based CQ Test
Parsing query...
Parsing clause SELECT: symbol, price, high, low
Parsing clause FROM: STOCK
Parsing clause WHERE:
Parsing clause GROUP BY:
Parsing clause ORDER BY: price
Parsing trigger condition...
Recognize TRIGGER TYPE: Time-based
Recognize TRIGGER CONDITION:
&&&&&&&&&0,1,2,3,4,5,6
Checking TRIGGER CONDITION: OK
Parsing stop condition...
Recognize STOP CONDITION: 6-6-1999 11:00:00
Passing the query to the data source...
Connecting to data source: 129.95.40.15
Executing query
Getting result from data source
Installing Continual Query on CQ server...
Connecting to CQ server: 129.95.40.15
Inserting an entry for Scheduling Manager
Updating user installed CQ list
Creating cache for the continual query
Now you can view the result.
Continual Query #186 was installed successfully!

Figure 6-14: CQ Installation and Execution Log Information

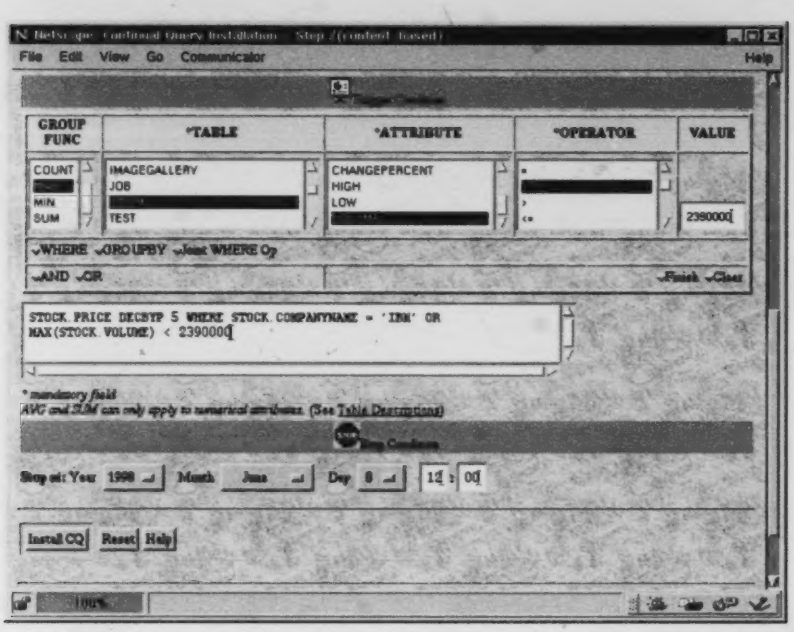


Figure 6-15: CQ Installation Step 2 (content-based) - Trigger and Stop Condition

Clear button

clears the content in the Trigger Condition Multiple Text Input Board, sets the trigger condition to *NULL*, and reset all the visual controls for the trigger condition input.

- *For the stop condition input*, the GUI controls are the same as those for time-based stop condition input which is described earlier in this section(in time-based CQ installation step 2).

Install CQ, Reset, and Help buttons have the same functions as those described earlier(in time-based CQ installation step 2).

6.4.2.4 Browse Installed CQ

- User Login Screen Every user can only view or delete his own continual queries. A *User Login Form* is provided for the access control. It is displayed in Figure 6-16.

- View Installed CQ Screen After the user clicks the *Submit* button on *User Login Form*, if the login information is valid, the user will see a similar page as shown in Figure 6-17. Then the user may view his installed CQs or delete some of them. The controls are described below:

  CQ ID text input box holds the single ID of the continual query the user wants to delete.

  Delete button will send the "Delete" command to the CQ server to delete a registered continual query whose ID appears in the *CQ ID* input box.

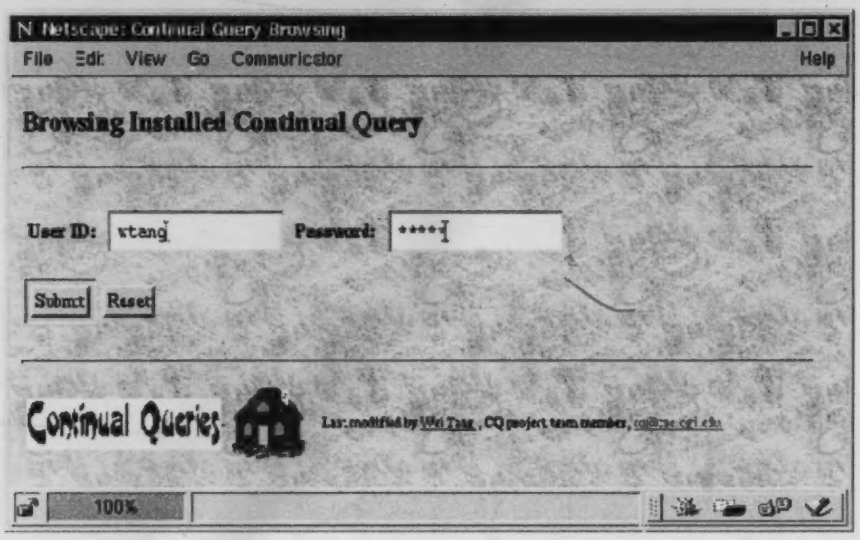


Figure 6-16: User Login Screen for Browsing CQ

Delete ALL button will submit the "Delete All" command to the CQ server to delete all the registered continual queries owned by the user.

6.4.3 Administration Services

For management and debugging purpose, the *CQ Services* include a set of *Administration Services* which can be used by the CQ system administrators.

6.4.3.1 Source Meta Query

The *Source Meta Query* is used to query over meta data of the CQ system. It is a service for the CQ system administrator, not for normal CQ users. The data password is required. *Source Meta Query Form* is similar to *Normal Query Entry Form*(See Section 6.2.1.3 for details). The difference is that the *FROM* control does not have the limitation of only the data tables. The administrator can select from all the meta tables in CQ system to check their contents. Figure 6-18 presents the source meta query screen.

6.4.3.2 Source Data Update

In order to simulate the data source update event, we provide this *Source Data Update service*. We then are able to observe and check the continual query evaluation behavior. First, we also need to login to the data source. The screen shows in Figure 6-19.

After logging into the data source, the *Table and Action Choice* page comes up. There are four visual components on the page: (1) the data table list; (2) *Insert* action button; (3) *Delete* action

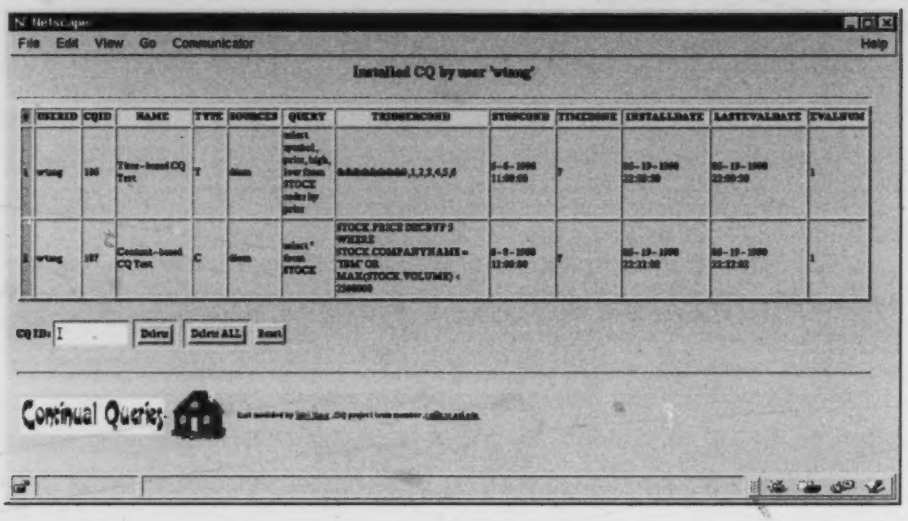


Figure 6-17: View Installed CQ Screen

button; (4) *Update* action button. The three buttons represent three database update operation types.

We take the *Update* button for example. After we choose the *Update* button on the *Table and Action Choice* page, the layout of the result page is shown in Figure 6-21 (suppose we chose the *BIB_SRC* table for update).

6.5 Further Discussions

The current CQ system prototype is still under development. New components are being introduced into the prototype. For example, the Weather Watch data source is available with time-based continual query facility. Bibliography CQ server will soon be operational online as well. Figure 6-22 and Figure 6-23 demonstrate both the data source front page and the query installation interface.

In the meantime, we are facing new technologies everyday, as well as emerging new Web standards, such as XML (Extensible Markup Language) and HTTP1.1 protocol. We are actively watching new technologies that can be incorporated into the current CQ prototype system.

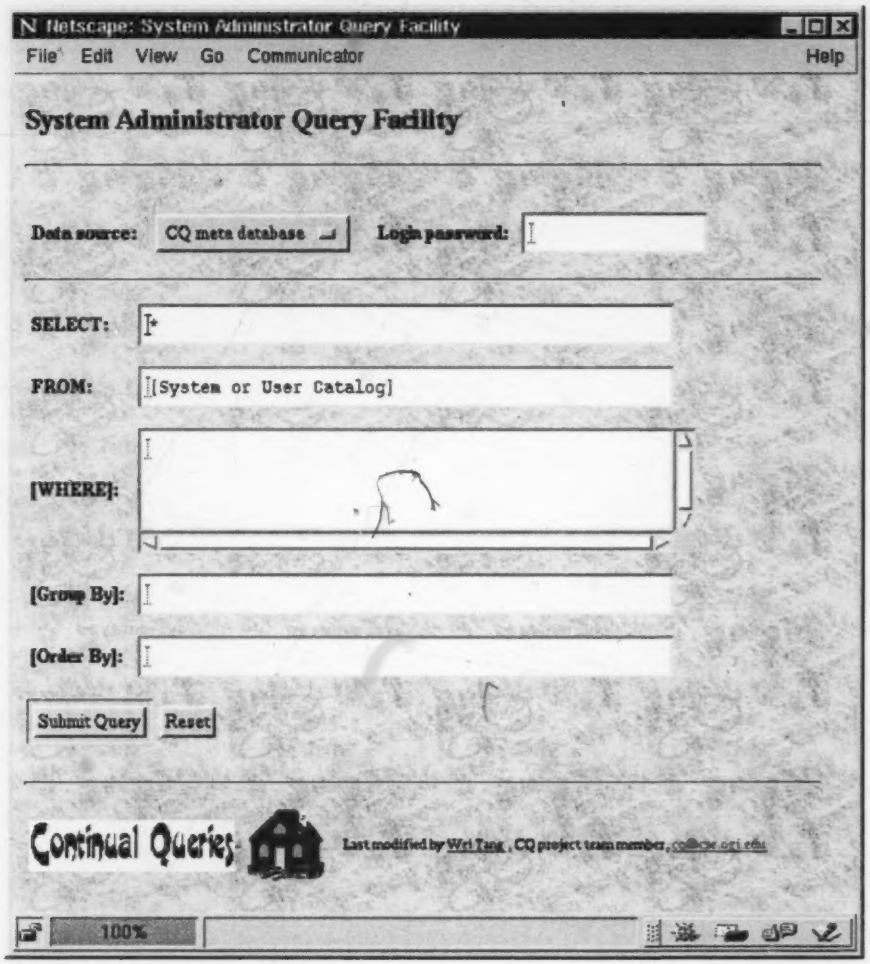


Figure 6-18: Source Meta Query Screen

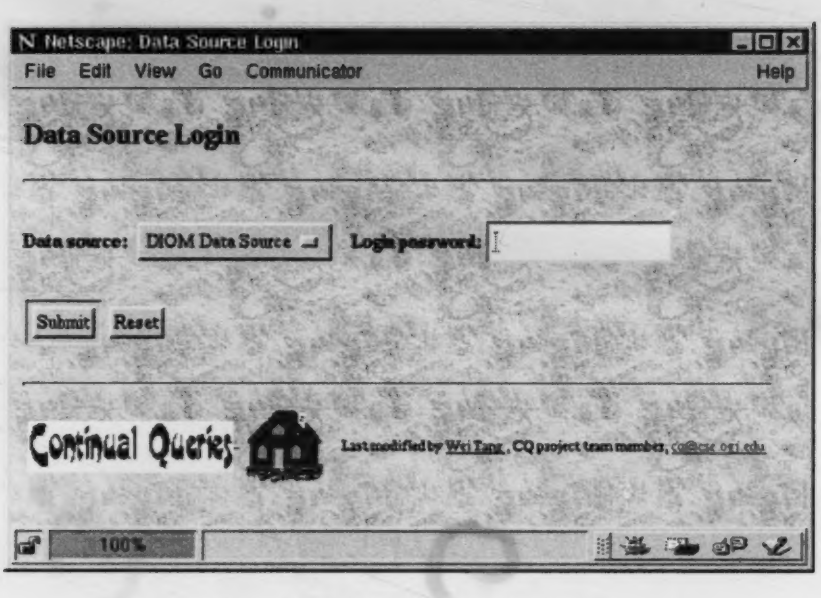

Netscape: Data Source Login
File Edit View Go Communicator Help
Data Source Login
Data source: DIOM Data Source
Login password:
Submit Reset
Continual Queries
Last modified by Wei Tang , CQ project team member, cq@cse.ogi.edu
100%

Figure 6-19: Data Source Login Screen

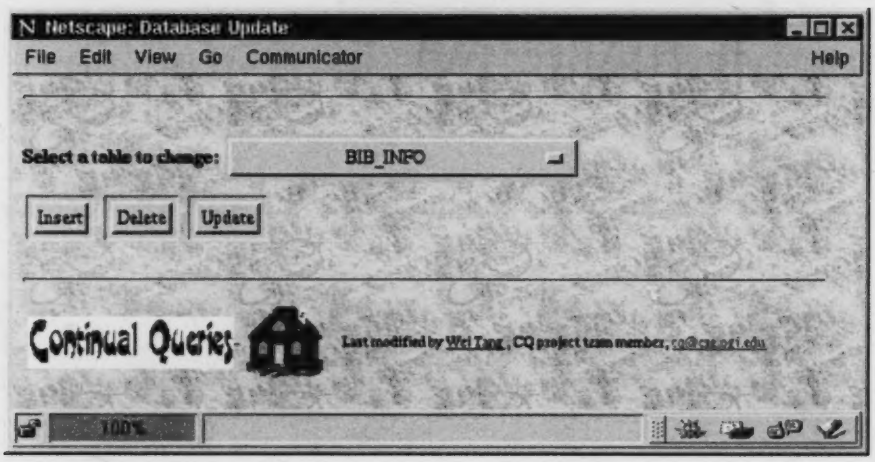

Netscape: Database Update
File Edit View Go Communicator Help
Select a table to change: BIB_INFO
Insert Delete Update
Continual Queries
Last modified by Wei Tang , CQ project team member, cq@cse.ogi.edu
100%

Figure 6-20: Table and Action Choice Screen

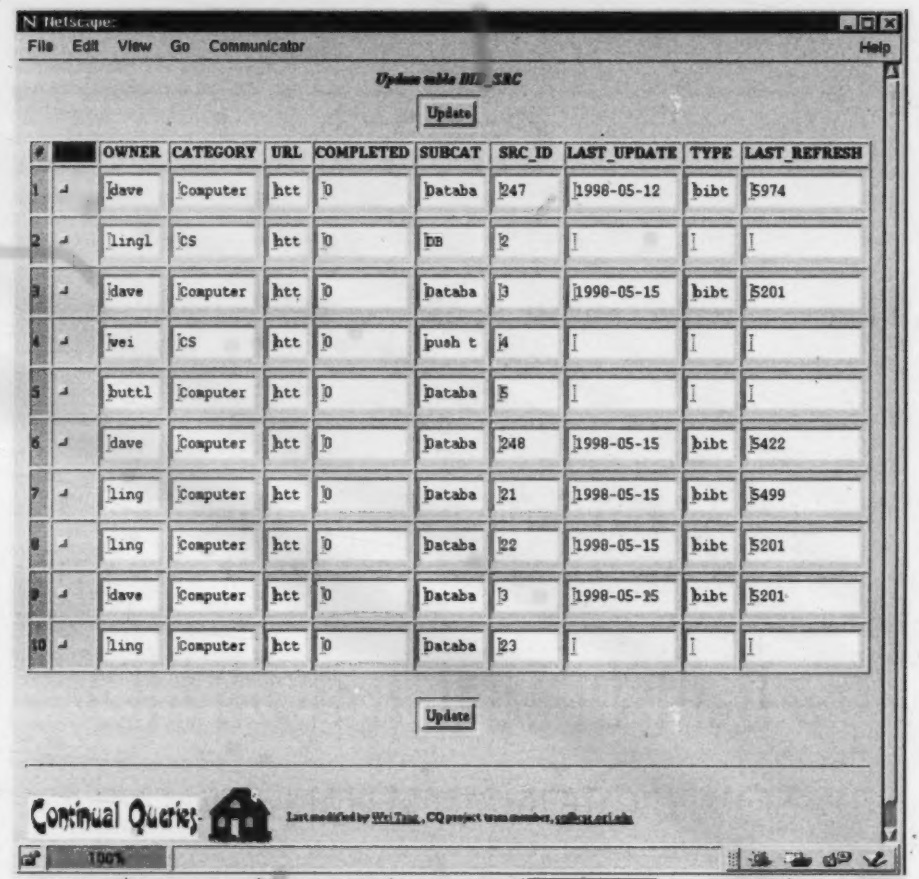

Netscape:
File Edit View Go Communicator Help
Update
OWNER CATEGORY URL COMPLETED SUBCAT SRC_ID LAST_UPDATE TYPE LAST_REFRESH
1 dave Computer htt 0 Databa 247 1998-05-12 bibt 5974
2 lingl CS htt 0 DB 2 I I I
3 dave Computer htt 0 Databa 3 1998-05-15 bibt 5201
4 wei CS htt 0 push t 4 I I I
5 buttl Computer htt 0 Databa 5 I I I
6 dave Computer htt 0 Databa 248 1998-05-15 bibt 5422
7 ling Computer htt 0 Databa 21 1998-05-15 bibt 5499
8 ling Computer htt 0 Databa 22 1998-05-15 bibt 5201
9 dave Computer htt 0 Databa 3 1998-05-25 bibt 5201
10 ling Computer htt 0 Databa 23 I I I
Update
Continual Queries
100%

Figure 6-21: Data Update Screen

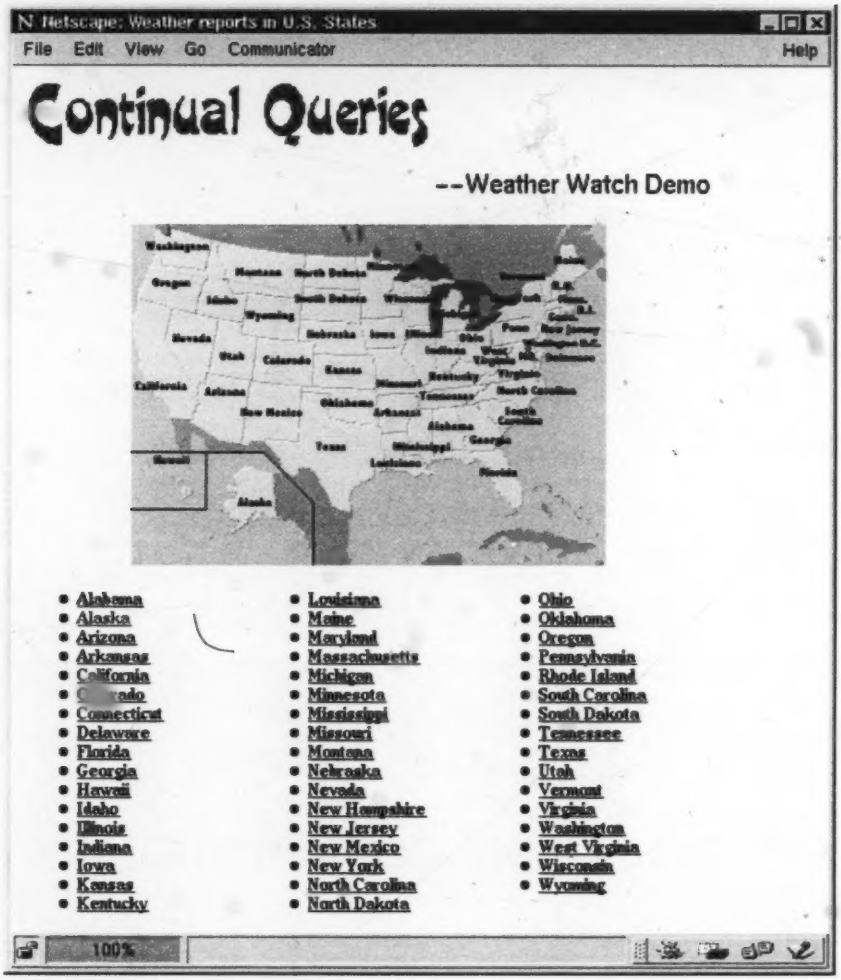


Figure 6-22: Continual Query Weather Watch source front page

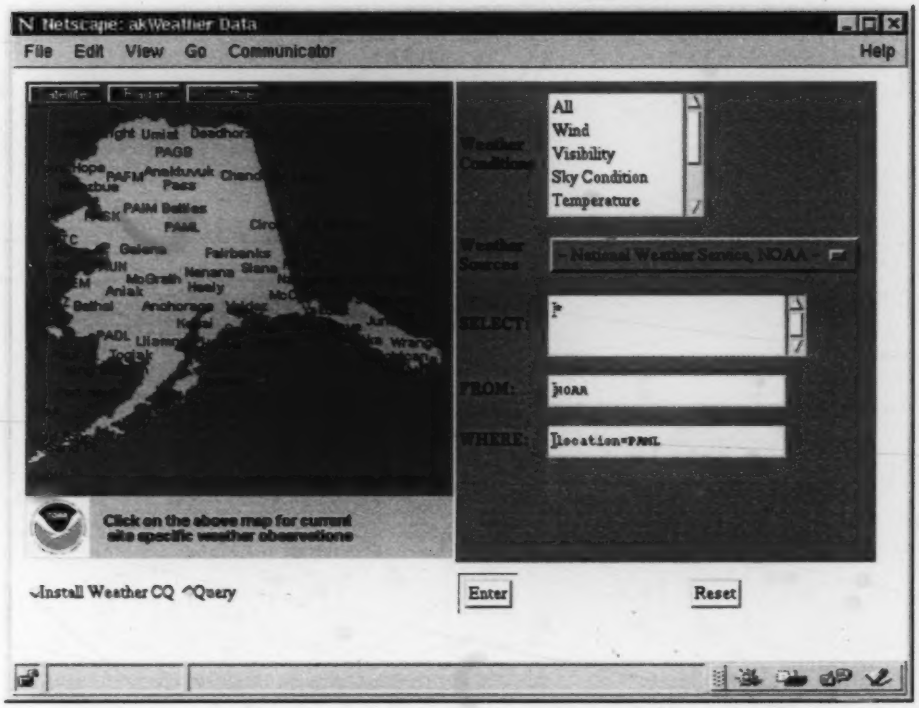


Figure 6-23: Continual Query Installation(query input) for Weather Watch data source

Chapter 7

Discussion and Related work

7.1 Pull, Push, and Continual Queries

Here we discuss the concept of pure pull data delivery and pure push data delivery, and compare them with the concept of continual queries and the event-driven data delivery using continual queries.

7.1.1 Overview of Data Delivery Protocols

Data delivery is defined as the process of delivering information from a set of information sources (servers) to a set of information consumers (clients). There are several possible ways that servers and clients communicate for delivering information to clients, such as clients request and servers respond, servers publish what are available and clients subscribe to only the information of interest, or servers disseminate information by broadcast. Each way can be considered as a protocol between servers and clients, and has pros and cons for delivering data in an open and dynamic information universe.

7.1.1.1 Client Request and Server Response

The *Request/Response* protocol follows the data delivery mechanism that clients send their request to servers to ask the information of their interest, servers respond to the requests of clients by delivering the information requested.

Current database servers and object repositories deliver data only to clients who explicitly request information from them. When a request is received at a server, the server locates or computes the information of interest and returns it to the client. The advantage of the *Request/Response* protocol is the high quality of data delivery since only the information that is explicitly requested by clients is delivered. In a system with a small number of servers and a very large number of clients, the *Request/Response* mechanism may be inadequate, because the server communication and data processing capacity must be divided among all of the clients. As the number of clients continuous to grow, servers may become overwhelmed and may respond with slow delivery or unexpected delay, or even refuse to accept additional connections.

7.1.1.2 Servers Publish and Clients Subscribe

The *Publish/Subscribe* protocol delivers information based on the principle that servers publish information online, and clients subscribe to the information of interest. Information delivery is primarily based on the selective subscription of clients to what is available at servers and the subsequent publishing from servers according to what is subscribed.

As the scale and rate of changes for online information continues to grow, the *Publish/Subscribe* mechanism attracts increasing popularity as a promising way of disseminating information over networks. Triggers and change notifications in active database systems bear some resemblance to the *Publish/Subscribe* protocol based on point-to-point communication [2]. The *Publish/Subscribe* mechanisms may not be beneficial when the interest of clients changes irregularly because in such situations clients may be continually interrupted to filter data that is not of interest to them. A typical example is the various online news groups. Another drawback is that publish/subscribe is mostly useful for delivering new or modified data to clients, but it cannot be used to efficiently deliver previously existing data to clients, which the clients later realize they need. Such data are most easily obtained through the request/respond protocol.

7.1.1.3 Servers Broadcast

The *Broadcast* mechanism delivers information to clients periodically. Clients who require access to a data item need to wait until the item appears. There are two typical types of broadcasting: *selective broadcast* (or so called *multicast*) and *random broadcast* [11]. Selective broadcast delivers data to a list of known clients and is typically implemented through a router that maintains the list of recipients. Random broadcast, on the other hand, sends information over a medium on which the set of clients who can listen is not known *a priori*. Note that the difference between selective broadcast and *Publish/Subscribe* is that the list of recipients in selective broadcast may change dynamically without explicit subscription from clients.

The *Broadcast* protocol allows multiple clients to receive the data sent by a data source. It is obvious that using broadcast is beneficial when multiple clients are interested in the same items. The tradeoffs of broadcast mechanisms depend upon the number of clients who have the commonality of interest and the volume of information that are of interest to a large number of clients [11, 2].

7.1.2 Overview of Data Delivery Modes

With the rapid growth of the volume and variety of information available online, combined with the constant increase of information consumers, it is no longer efficient to use a single mode of data delivery. A large-scale modern information system must provide adequate support for different modes of data delivery in order to effectively cope with the various types of communications between clients and servers to improve query responsiveness. Another benefit of providing different modes of data delivery is to allow the system to be optimized for various criteria according to different

requirements of data delivery. In this section we identify three potentially popular modes of data delivery and compare them with the types of delivery protocols that can be used. They are client pull-only option, server push-only option, and client pull with server push combined option.

7.1.2.1 Pull-only Mode

In the *Pull-only* mode of data delivery, the transfer of data from servers to clients is initiated by a client pull. When a client request is received at a server, the server responds to it by locating the requested information. The *Request/Respond* style of client and server communication is *pull-based.*

The main characteristic of pull-based delivery is that the arrival of new data items or updates to existing data items are carried out at a server without notification to clients unless clients explicitly poll the server. Also, in pull-based mode, servers must be interrupted continuously to deal with requests from clients. Furthermore, the information that clients can obtain from a server is limited to when and what clients know to ask for. Conventional database systems (including. relational and object-oriented database servers) and most of the web search engines offer primarily pull-based data delivery.

7.1.2.2 Push-only Mode

In *Push-only* mode of data delivery, the transfer of data from servers to clients is initiated by a server push in the absence of specific request from clients. The main difficulty of push-based approach is to decide which data would be of common interest, and when to send them to clients (periodically, irregularly, or conditionally). Thus, the usefulness of server push depends heavily on the accuracy of a server to predict the needs of clients. *Broadcast* style of client and server communication is a typical *push-only* type.

In push-based mode, servers disseminate information to either an unbounded set of clients (random broadcast) who can listen to a medium or a selective set of clients (multicast) who belong to some categories of recipients that may receive the data. It is obvious that the push-based data delivery avoids the disadvantages identified for client-pull approaches such as unnoticed changes. A serious problem with push-only style, however, is the fact that in the absence of a client request the servers may not deliver the data of interest in a timely fashion. A practical solution to this problem is to allow the clients to provide a profile of their interests to the servers. The *Publish/Subscribe* protocol is one of the popular mechanisms for providing such profiles. Using publish/subscribe, clients (information consumers) subscribe to a subset of a given class of information by providing a set of expressions that describe the data of interest. These subscriptions form a profile. When new data items are created or existing ones are updated, the servers (information providers) publish the updated information to the subscribers whose profiles match the items.

7.1.2.3 Hybrid Mode

The hybrid mode of data delivery combines the client-pull and server-push mechanisms. The continual query approach [25] presents one possible way of combining the pull and push modes, namely, the transfer of information from servers to clients is first initiated by a client pull and the subsequent transfer of updated information to clients is initiated by a server push.

The hybrid mode represented by continual queries approach can be seen as a specialization of push-only mode. The main difference between hybrid mode and push-only mode is the initiation of the first data delivery. More concretely, in a hybrid mode, clients receive the information that matches their profiles from servers continuously. In addition to new data items and updates, previously existing data that match the profile of a client who initially pull the server are delivered to the client immediately after the initial pull. However, in push-only mode, although new data and updates are delivered to clients with matching profiles, the delivery of previously existing data to clients who subsequently realize that they need it is much more difficult than through a client pull.

7.1.3 Pure Push versus Continual Queries

In a pure bush environment such as broadcast services, the server broadcast the update periodically and the clients may tune the channels to listen to those broadcast information that is of particular interest to them. Thus, the data is pushed from source to the broadcast server and then pushed from the server to the client. Figure 7-1 shows the typical data delivery flow in a pure push environment.

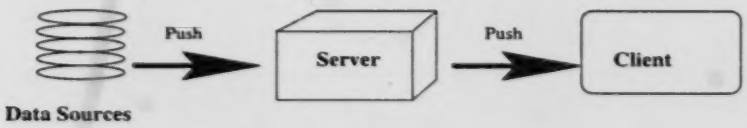


Figure 7-1: The data delivery flow in a broadcast-based push service

Continual Queries server is not a pure-push based server since the request is initiated by a client pull. Once the client pulled the CQ server at the time of installing a CQ, the CQ server starts pushing the subsequent updates that satisfy the update threshold specified in the CQ to the client continually until the termination condition is met. The data delivery flow is shown in Fig. 7-2.

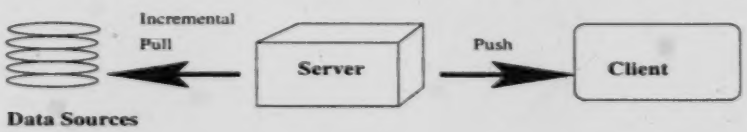


Figure 7-2: The data delivery flow in a broadcast-based push service

7.2 Related Work on DB Areas

The concept of continual queries was motivated and evolved by the increasing demand on event-driven information delivery. It was also inspired by the work on continuous queries by Terry et alia [37] at the early stage of the development. Comparing with Terry et al [37]'s proposal, there are a number of functionality differences. First, their proposal made several assumptions that seriously restricted the applicability of their technique to the Internet. Perhaps the most significant assumption is the limitation of database updates to append-only, disallowing deletions and modifications. Since this assumption is used in their query transformation algorithm, it has been difficult to relax it [4], when following their definition of continuous queries. Second, the specification model for update monitoring is purely time-based. There is no clean separation of query and trigger condition in the specification.

In addition, there has been considerable research done in the monitoring of information changes in databases. Powerful database techniques such as active databases and materialized views have been developed. The design of the CQ system is mostly inspired by the research in these fields. The following discussion should not be seen as a critique of these techniques. Rather, these techniques have been proposed primarily for "data-centric" environments, where data is well organized and controlled. When applied to an open information universe as the Internet, these assumptions no longer hold (see [21] for a summary of desired system properties in the Internet), and some of the techniques do not easily extend to scale up to the distributed interoperable environment.

7.2.1 Active Databases

Most of active database systems [38] provide facilities [6, 30, 35] that allow users to specify, in the form of rules, actions to be performed following changes of database state. Despite their conceptual generality, rules have been so far supported in a fairly restrictive form in practical systems, for example, by built-in triggers [15] in relational database management systems such as Oracle, Sybase, and Informix (see a further discussion On Commercial database trigger below). Active queries, introduced in Alert [35], is yet another form of ECA rules. Active queries are more sophisticated than database triggers, since they can be defined on multiple tables, on views, and can be nested within other active queries. However, active queries heavily rely on the use of active tables as system built-in capability and a number of concrete extensions to a particular system – IBM Starburst DBMS [13].

7.2.2 Materialized Views

Materialized views store a snapshot of selected database state. When a database is updated, the materialized view must be refreshed to reflect the updates. A naive solution is to rematerialize the view from the base data. In contrast, incremental update algorithms are believed to carry lower execution cost if changes to the database are moderate [14, 19]. Three approaches have been

described previously. The first approach refreshes the view immediately after each update to the base table [5]. The second defers the refresh until a query is issued against the view [34]. The third refreshes the view periodically [19]. The main tradeoff in choosing among these approaches is the staleness of the view data vs. the cost of updating it. Most of the algorithms in the literature [5, 14, 12] work in a centralized database environment, in which the materialized view and its base tables co-reside. The study on distributed materialized view management has been primarily focused on determining the optimal refresh sources and timing for multiple views defined on the same base data [36]. Other works on distributed environments include quasi-copies for replication [3] and update anomalies in data warehouses [40]

7.2.3 Commercial database triggers

Conceptually a database (built-in) trigger is an event–condition–action (ECA) rule in a restrictive form. Commercial DBMSs have been introducing support for database triggers at various levels, mainly due to the customers' need for better support for integrity constraints. In the SQL standard, checking of constraints, such as `price > 0` or referential integrity constraints, is triggered by the DBMS. Users can specify whether constraints are to be checked at the end of each SQL statement. However, support for triggers in SQL standard is limited. The trigger events can only be built-in SQL operations (update, insert, delete) on a single base table. The triggers can be specified only on a single base table. Triggers over views are not allowed. Database triggers can only be part of the triggering transactions and triggers can not be nested. For instance, Unlike continual queries, Sybase allows only one trigger to be associated with an operation on a table. The action part of the trigger is limited to a sequence of SQL statements. Further, triggering is restricted to one level where triggered actions themselves do not cause triggers to be fired.

7.3 Related Work in Web-based Systems

The most distinct features of the World Wide Web is the vast amount of information published online and the fact that the information on the Web may change at any time and in any way. These rapid and often unpredictable changes to the information sources create a new problem: how to detect, represent, and notify these changes.

In addition to the Continual Queries project, there are several systems developed towards monitoring source data changes. One type of systems are the extension of Web search engines or search software by monitoring the URL changes and notifying the users when the URLs of the data sources of interest have changed. A representative system is the URL-Minder (`http://www.netmind.com/`. Another interesting web page change notification tool is the Web-site News monitor (by AT&T at `http://www.research.att.com/ chen/web-demo/`. It allows users to watch some selected web site pages for changes from a small and fixed selection of Web sites. Another type of projects is the change detection over HTML pages, such as the *C*3 [7] project at Stanford. (http://www-

db.stanford.edu/c3/c3.html) It addresses the problem of change management by using change detection mechanisms such as fetching pages from the source sites and making difference with the previous results. Another interesting feature of $C3$ is to allow users to query over the change databases and provide query subscription service to allow users to subscribe to the change notification system such that the change will be notified by the users. Another way is to use polling and allow users to learn about changes in a number of ways: (1) whenever changes of certain kind occurs, (2) by weekly or daily report, or (3) by explicit request only.

The main difference between the continual queries systems and the domain of applications is the its event-driven nature and its support of polling and filtering based on the users application requests. For example, the CQ system is able to install a continual query *"Notify me whenever IBM and AT&T each drops more than 5%"*.

Chapter 8

Conclusion and Future Work

8.1 Summary and Conclusion

The thesis describes the design and implementation of a prototype system for *data update monitoring* in a distributed open environment (such as the Internet). It uses the *continual query* concept as a powerful means for supporting continual monitoring of information updates. The work presented here is built on top of previous work in *Continual Queries* [25, 24] as well in DIOM [26]. The main contribution of this thesis can be summarized as follows:

- We have defined the *continual query* specification language and implemented a subset of the continual query language syntax.
- We have experimented and realized the *continual query* concept in a working distributed system and proved the power of the continual query concept in supporting event-driven update monitoring.
- We have designed and implemented the first prototype system architecture which incorporates *DIOM* distributed query scheduling framework, and have built the first version of a working system using several software tools such as Perl programming language, HTML, and JavaScript through CGI and HTTPD1.0.

The most attractive features of the *Continual Query* prototype system implementation include

- a graphical Web interface offering user registration service, online queries over heterogeneous data sources, *continual query* installation and other client services, as well as system administration tools;
- a system meta data structure for managing and maintaining system and application metadata;
- a relational database (Oracle) wrapper with reusable modules such as automatic trigger installation module, system-controlled polling module;

- a suite of *Continual Query* server kernel modules, including *Object Manager*, *Event Detector*, *Trigger Condition Evaluator*, *Continual Query Evaluator using naive-algorithm*, and *Email-based Change Notification Manager*;
- a design and realization of the integration of CQ server kernel modules with other continual query system components developed by other team members such as the Weather-Watch source wrapper and the bibliography source wrapper, thus providing *Continual Query services* to applications in diverse domains.

8.2 Future Development

As mentioned in previous chapters, the first prototype of the *Continual Query* system has made explicit simplification on the proposed *Continual Query* system architecture in several aspects. First, it does not solve all the update monitoring problems in a distributed environment. For example, the applicability of the current implementation relies on the construction of CQ wrappers in order to be used for any new information sources. The techniques used in the current prototype for building wrappers are not generic enough to allow rapid generation of CQ wrappers to any given type of new information sources. In order to make the implementation close to a production-quality system, more research work and implementation steps need to be carried out. However, this first CQ prototype system will be used as a testbed for further development and experiment of the continual query concept and algorithms as well as the study of general infrastructure for event-driven update monitoring in a distributed open environment. Here we outline some directions of future development of the system:

- The first aspect to be enhanced is the GUI components to provide users with more functionality and flexibility, and make them easier to use. We intent to use Java applets at client side instead of the current combined use of CGI and JavaScript.
- We would like to experiment with JDBC to access database sources instead of Perl DBI:DBD modules to allow the implementation more interoperable with external systems.
- We plan to replace all the CGI scripts with Java servlets to improve the overall system performance, which may include the design and implementation of an extensible and reusable Java servlet package
- We would also like to enrich the continual query semantics and syntax to include richer event expressions, offer more flexibility in termination condition specification and means for notification.
- We plan to design a full fledged object model, event model, observation (event detection) model and notification model in the next generation of the *continual query* system.

- We are interested in studying various algorithms and strategies for continual query evaluation optimization and cache management optimization.
- We plan to incorporate the DRA in continual query evaluation manager and furthermore to conduct performance evaluation of continual query evaluation algorithms.

We believe that the *Continual Query* system presents an interesting architecture for Internet scale event-driven information delivery.

Personally I have learned a lot through research and hand-on experience with the CQ prototype system development, and have had a lot of fun in addition to hard work. It was a real good feeling seeing the prototype running on the Web, especially seeing the smooth integration of the CQ server kernel modules with the other CQ wrappers that are developed independently.

Bibliography

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[4] D. Barbara and R. Alonso. Processing continuous queries in general environments. Technical report, Matsushita Information Technology Laboratory, Princeton, NJ, June 1993.

[5] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 61–71, Washington, DC, May 1986.

[6] S. Chakravarthy. Architectures and monitoring techniques for active databases: An evaluation. In *Technical Report TR-92-041, University of Florida*, Gainesville, FL, 1992.

[7] S. Chawathe, S. Abiteboul, and J. Widom. Managing and querying changes in seminstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[8] P. Cohen, A. Cheyer, M. Wang, and S. Baeg. An open agent architecture. In *AAAI Spring Symposium*, pages 1–8, March 1994.

[9] P. Cohen and H. Levesque. Communicative actions for artificial agents. In *Proceedings of the International Conference on Multi-Agent Systems, AAAI Press*, June 1995.

[10] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[11] M. Franklin and S. Zdonik. Dissemination-based information systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):20–30, September 1996.

[12] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 157–166, Washington, DC, May 1993.

[13] L. Haasi, W. Chang, G. Lohman, J. McPherson, P.Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 377–388, March 1990.

[14] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 440–453, San Francisco, CA, May 1987.

[15] Informix Software, Inc. *Informix Guide to SQL: Syntax (Version 6.0)*, 1994.

[16] B. Kahler and O. Risnes. Extending logging for database snapshot refresh. In *Proceedings of the International Conference on Very Large Data Bases*, pages 389–398, Brighton, England, September 1987.

[17] Y. Lee. Rainbow: A prototype of the diom interoperable system. MSc. Thesis, Department of Computer Science, University of Alberta, July, 1996.

[18] Y.-S. Lee. Rainbow: Prototyping the diom interoperable system(tr96-32). Technical report, Department of Computer Science, University of Alberta, Edmonton, Alberta, Fall 1996.

[19] B. Lindsay, L. Haas, and C. Mohan. A snapshot differential refresh algorithm. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 53–60, Washington, DC, May 1986.

[20] L. Liu and C. Pu. The diom approach to large-scale interoperable information systems. Technical report, TR95-16, Department of Computing Science, University of Alberta, Edmonton, Alberta, March 1995.

[21] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. In *ACM International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, USA, November 1995.

[22] L. Liu and C. Pu. A dynamic query scheduling framework for distributed and evolving information systems. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Baltimore, May 27-30 1997.

[23] L. Liu and C. Pu. Complex event specification and event detection for continual queries. Technical report, OGI/CSE, Portland, OR, March 1998.

[24] L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. Technical Report TR-95-17, Department of Computer Science, University of Alberta, July 1995.

[25] L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 27-30 1996.

[26] L. Liu, C. Pu, and Y. Lee. Adaptive approach to query mediation across heterogeneous information sources. In *International Conference on Cooperative Information Systems(CoopIS)*, Brussels, Belgium, June 1996.

[27] L. Liu, C. Pu, W. Tang, J. Biggs, D. Buttler, W. Han, P. Benninghoff, and Fenghua. Cq: A personalized update monitoring toolkit. In *SIGMOD'98*, 1998.

[28] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[29] T. Maude and GrahamVillis. *Rapid Prototyping: the Management of Software risk*. Pitman Publishing, 1991.

[30] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 215–224, May 1989.

[31] R. S. Pressman. *Software Engineering: a Practitioner's Approach 3rd Edition*. McGraw-Hill, 1992.

[32] K. Richine. Distributed query scheduling in the context of diom: An experiment. MSc. Thesis, Department of Computer Science, University of Alberta, April, 1997.

[33] A. Rosenthal and U. Chakarvarthy. Anatomy of a modular multiple query optimizer. In *The International Conference on Very Large Data Bases*, 1988.

[34] N. Roussopoulos and H. Kang. Preliminary design of adms+: A workstation-mainframe integrated architecture for database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 355–364, Kyoto, Japan, August 1986.

[35] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings of the International Conference on Very Large Data Bases*, pages 469–478, Barcelona, Spain, September 1991.

[36] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the 6th International Conference on Data Engineering*, pages 512–520, Los Alamitos, February 1990.

[37] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, CA, January 1992.

[38] J. Widom and S. Ceri. *Active Datanase Systems.* Morgan Kaufmann, 1996.

[39] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, March 1992.

[40] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

Appendix A

Continual Query Syntax

The syntax below is just a subset of the **Continual Query** *syntax. The first Continual Query prototype system implementation is based on this simplified syntax.*

```
<CQ> ::= <Query> <TriggerCond> <StopCond>

<Query> ::= SELECT <SelectList>
   FROM <ObjectList>
   [WHERE <SearchCondition>]
   [GROUP BY <AttributeList>]
   [ORDER BY <SortSpecList>]

<TriggerCond> ::= <TimeTriCond> | <ContentTriCond>

<StopCond> ::=  <Month> '-' <Day> '-' <Year> ' '
                <Hour> ':' <Min> ' ' <TimeZone>

<SelectList> ::= * | <AttributeList>

<AttributeList>:= <Attribute> | <Attribute> [, <AttributeList>]

<Attribute> ::= id | <ObjectName>.<AttributeName> | <AggreSpec>

<ObjectName> ::= String

<AggreSpec> ::= COUNT(*) | <AggreFunc>(<Attribute>)

<AggreFunc> ::= AVG | MAX | MIN | SUM | COUNT

<ObjectList> ::= <ObjectName> | <ObjectName> [, <ObjectList>]

<SearchCondition> ::= <BoolExpr> | [NOT] <BoolExpr>

<BoolExpr> ::= <BoolTerm> | <BoolTerm> <LogicOp> <BoolExpr>

<LogicOp> ::= AND | OR

<BoolTerm> ::= <Predicate> | <ValueExpr>

<Predicate> ::= <ValueExpr> <Op> <ValueExpr>

<Op> ::= <CompOp> | [NOT] <LikeOp>

<CompOp> ::= <> | = | < | > | <= | >=

<LikeOp> ::= LIKE

<ValueExpr> ::= <NumValExpr> | <StrValExpr>
```

```
<NumValExpr> ::= <Term> | <NumValExpr> + <Term> | <NumValExpr> - <Term>

<Term> ::= <Factor> | <Term> * <Factor> | <Term> / <Factor>

<Factor> ::= [+|-] | <NumValue>

<NumValue> ::= number | <Attribute> | (<NumValExpr>)

<StrValExpr> ::= <StrValue>

<StrValue> ::= String | <Attribute>

<SortSpecList> ::= <SortSpec> | <SortSpec> [, <SortSpec> ]

<SortSpec> ::= <Attribute> [<OrderKey>]

<OrderKey> ::= ASC | DESC

<TimeTriCond> ::= <MinExpr> '&&' <HourExpr> '&&' <DayOfMonExpr> '&&'
                  <MonthExpr> '&&' <DayOfWeekExpr>

<MinExpr> ::= <MinFactor> | <MinFactor> [, <MinExpr>] | <NotSpecified>

<MinFactor> ::= <MinVal> | <MinVal> <To> <MinVal>

<NotSpecified> ::= null

<MinVal> ::= [0-5][0-9]

<To> ::= -

<HourExpr> ::= <HourFactor> | <HourFactor> [, <HourExpr>] | <NotSpecified>

<HourFactor> ::= <HourVal> | <HourVal> <To> <HourVal>

<HourVal> ::= [0-1][0-9] | [2][0-3]

<DayOfMonExpr> ::= <DayOfMonFactor> | <DayOfMonFactor> [, <DayOfMonExpr>]
                   | <NotSpecified>

<DayOfMonFactor> ::= <DayOfMonVal> | <DayOfMonVal> <To> <DayOfMonVal>

<DayOfMonVal> ::= [0-2][0-9] | [3][0-1]

<MonthExpr> ::= <MonthFactor> | <MonthFactor> [, <MonthExpr>] | <NotSpecified>

<MonthFactor> ::= <MonthVal> | <MonthVal> <To> <MonthVal>

<MonthVal> ::= [0-9] | [1][0-1]

<DayOfWeekExpr> ::= <DayOfWeekFactor> | <DayOfWeekFactor> [, <DayOfWeekExpr>]
                    | <NotSpecified>

<DayOfWeekFactor> ::= <DayOfWeekVal> | <DayOfWeekVal> <To> <DayOfWeekVal>

<DayOfWeekVal> ::= [0-6]

<ContentTriCond> ::= <ContTriGroup> | <ContTriGroup> <EventOp> <ContentTriCond>

<ContTriGroup>  ::= <ContPrimitive> <GrpConstraint>

<ContPrimitive> ::= [<AggreFunc>(]<ObjectName>.<Attribute>[)]
                    <ContTriCondOp> [ <Value> ]
```

```
<GrpConstraint> ::= WHERE <ContPrimitiveList> [GROUPBY <AttributeList>]

<ContPrimitiveList> ::= <ContPrimitive> | <ContPrimitiveList>
                        <GrpJointOp> <ContPrimitive>

<EventOp> ::= AND | OR | <sequence> | <parallel>

<ContTriCondOp> ::= <> | = | < | > | <= | >= | CHANGES | CONTAINS | LIKE
                  | INCBY | DECBY | INCBYP | DECBYP

<GrpJointOp> ::= AND | OR

<ContTriVal> ::= String

<Year> ::= [0-9][0-9][0-9][0-9]

<Month> ::= <MonthVal>

<Day> ::= <DayOfMonVal>

<Hour> ::= <HourVal>

<Min> ::= <MinVal>

<TimeZone> ::= [-|+][0-1][0-9]
```